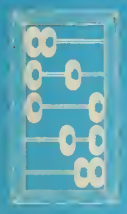
UIUCDCS-R-78-922 UILU-ENG 78 1747

MINIMUM MEAN RUNNING TIME FUNCTION GENERATION USING READ ONLY MEMORY

BY

GILLES HENRI GARCIA

December 1978



DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS

MINIMUM MEAN RUNNING TIME FUNCTION GENERATION USING READ ONLY MEMORY

BY

Gilles Henri Garcia

Ing., Ecole Superieure d'Electricite, 1972
M.S., University of Illinois, 1976

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1979

Urbana, Illinois

A ma famille

MINIMUM MEAN RUNNING TIME FUNCTION GENERATION USING READ ONLY MEMORY

Gilles Henri Garcia, Ph.D.
Department of Computer Science
University of Illinois at Urbana-Champaign, 1979

This thesis presents a method for generating functions using a minimum mean running time polynomial approximation. Although the method can be used for software library routines, the work focuses on hardware implementations. With the advent of VLSI, Read Only Memories may be used to store many constants very economically. In addition, large multipliers are now available which multiply two n-bit numbers in O(n) or O(logn) time. In this discussion, multiplication is considered as the basic operator. The method consists of splitting the interval [a,b] into several large partitions. Within each large partition, the function f(x) is evaluated in a large number of small subintervals using an approximating polynomial of very low degree. The coefficients of the polynomials are stored in ROMs. A set of partitions of [a,b] is defined such that the polynomials have the same degree within each partition. The optimal determination of the partition is obtained by solving a non-linear programming problem where the objective is the minimization of the average number of multiplications over the whole range, assuming a logarithmic distribution of

of the mantissas of floating-point numbers, and the cost constraint is the maximum number of ROM words available. A program has been written to solve this NLP problem and a set of optimal curves representing the average number of multiplications achievable vs. the number of ROM words available has been obtained for the function $f(x) = 1/x$. All possible combinations of 2 and 3 partitions using Chebyshev polynomials of degrees 1 to 5 were considered. Various bounds are given on the complexity of the method. It can generate $f(x)$ to precision n in $O(\log(\frac{n}{\gamma \log n})M(n))$ time, where $M(n)$ is the time to multiply two n-bit integers and γ is a positive (non-zero) constant. The method uses $O(n^{\gamma + 2})$ words of memory and $O(\frac{n}{\log n})$ processors.

ACKNOWLEDGMENTS

I would like to thank my thesis advisor Professor William J. Kubitz for the advice, guidance and beyond-the-call-of-duty encouragement he gave me during these interminable years.

I am particularly grateful to Professor A. Sameh for his interest and helpful suggestions, and to the other members of my thesis committee, Professors W. J. Poppelbaum, J. E. Robertson, and E. H. Davidson. In addition, I would like to thank the Department of Computer Science for their financial support.

I wish to thank Lucy van Weringh for her patience, typing and psychological counseling, which had to be reciprocated at times.

I am also indebted to Gayanne Carpenter and to Cinda Robbins for their assistance. And finally, I am especially indebted to Dan Pitt with whom I have had many fruitful conversations about other things.

TABLE OF CONTENTS

Page

1. INTRODUCTION 1

1.1 Sequential vs. Parallel Function Generation Methods 5

1.2 Current Parallel Methods 7

1.2.1 Stefanelli's Inverting Circuit 9

1.2.2 Chin's Method 12

1.3 Rationale for the Approach Used Here 14

2. POLYNOMIAL APPROXIMATIONS FOR HARDWARE FUNCTION GENERATION 16

2.1 Choice of a Criterion for Goodness of Fit 18

2.2 Colloquation Points 22

2.3 Quasi-Optimal Polynomial Approximation 24

2.4 Piecewise Polynomial Approximation 25

2.5 The Logarithmic Distribution of Leading Digits in Floating-Point Numbers 27

2.6 Notation Used 29

3. ARCHITECTURE FOR HIGH-SPEED FUNCTION GENERATION: FOUR CLASSES 32

3.1 Uniform Interval Length Using Uniform Degree Polynomials (UIUD) 33

3.2 Variable Interval Length Using Uniform Degree Polynomials (VIUD) 35

3.3 Variable Degree Polynomials with either Uniform or Variable Intervals (UIVD and VIVD). 40

3.3.1 The Minimum Mean Running Time Approximation 42

3.3.2 Algorithm for Finding the Number of ROM Words 46

3.3.3 Linearization of the Non-Linear Programming Problem 60

4. THE SOLUTION OF THE NON-LINEAR PROGRAMMING PROBLEM FOR f(x) = 1/x 65

4.1 Penalty Function Method 66

4.2 Software Considerations 72

4.3 Study of the Minimum 81

4.3.1 Influence of μ 89

4.3.2 Influence of the Maximum Number of ROM Words R_O 89

5. RESULTS AND EXAMPLES 101

5.1 Average Number of Multiplications vs. Number of ROM Words: Optimal Curves 101

5.1.1 One Joint Results 103

5.1.2 Two Joint Results 118

5.2 The Effect of a Finite-Length Look-Up Address . 130

6. HARDWARE IMPLEMENTATION 132

6.1 Design Considerations: Expected Time Delay. How to Store the Coefficients 132

6.2 Architectural Considerations: Obtaining the Address from the Argument 139

6.2.1 UIUD Case 139

6.2.2 VIUD Case 142

6.2.3 UIVD Case 149

6.2.4 VIVD Case 152

7. THE COMPLEXITY OF THE METHOD 154

7.1 Introduction and Background 154

7.1.1 The Measure of Effectiveness of an Algorithm 155

7.1.2 The Best Known Bounds for Parallel Polynomial Evaluation 156

7.1.3 Notation and Assumptions 157

7.1.4 Time vs space. An Intuitive Look at the R-Method 161

7.2 Memory Complexity of the R-Method 164

7.2.1 Defintion of Type 1 and Type 2 Functions 165

7.2.2 Memory Complexity of the Method 166

7.2.3 The Relationship between Polynomial Degree and ROM Size 169

7.3 The Effectiveness of the R-Method 177

7.3.1 The S-Method as a Particular Case of the R-Method 177

7.3.2 The Number of Processors, Speed-Up, Cost, and Efficiency for the R-Method ..181

7.3.3 Comparisons Using 1 Processor 184

7.3.4 Comparisons Using p Processors 188

7.3.5 Comparisons to the Best Known Bound for Function Evaluation 191

8. CONCLUSIONS 192

8.1 Areas of Further Research 192

8.2 Summary and Conclusions 198

LIST OF REFERENCES.....200

APPENDIX.....205

A.1 Program Listings.....205

A.2 Distribution of Floating Point Numbers Outside the Interval $[1/\beta, 1]$ 219

A.3 Limit on ROM Savings Coefficient When $n \to \infty$ 222

VITA 227

LIST OF FIGURES

Page

1.1 Stefanelli's Generation of 1/x 11

2.1 Colloquation Points 23

2.2 Notation Used Throughout the Discussion 30

3.1 Uniform Interval Length Case 34

3.2 UIUD Design Procedure 36

3.3 Illustration of $C(f, P_m, x_i, x_i + h)$ as a Function of x_i for $f = \frac{1}{x}$ 37

3.4 VIUD Design Procedure 39

3.5 Variable Degree Polynomials with either Uniform or Variable Intervals (UIVD and VIVD) ... 41

3.6 Number of Breakpoints in the Non-Uniform Case ... 52

3.7 Number of Breakpoints in the Non-Uniform Case; Different Behavior of $N(x_o,x)$ 54

3.8 Number of Breakpoints in the Non-Uniform Case; Another Behavior of $N(x_o,x)$ 55

4.1 Computer Solution; Subroutine Structure 74

4.2 Average Number of Multiplications; No Penalty ... 83

4.3 Number of ROM Words (View 1) 84

4.4 Number of ROM Words (View 2, Rotation) 85

4.5 Number of ROM Words (View 3, Rotation) 86

4.6 Location of the Minimum on the Surface Representing Objective + Penalty as a Function of Joints Locations 87

4.7 Combination of Average Number of Multiplications and the Penalty on the Number of ROM Words 90

4.8 Influence of μ; $\mu = 0$ 91

4.9 Influence of μ; $\mu = 10$ 92

4.10 Influence of μ; $\mu = 100$ 93

4.11 Influence of μ; $\mu = 10{,}000$ 94

4.12 Influence of μ; $\mu = 1{,}000{,}000$ 95

4.13 Influence of the Maximum Number of ROM Words R_o on the Position of the Minimum; $R_o = 200$ 97

4.14 Influence of the Maximum Number of ROM Words R_o on the Position of the Minimum; $R_o = 350$ 98

4.15 Influence of the Maximum Number of ROM Words R_o on the Position of the Minimum; $R_o = 500$ 99

4.16 Influence of the Maximum Number of ROM Words R_o on the Position of the Minimum; $R_o = 1024$ ...100

5.1 Each Point on the Curve Represents a Minimum 104

5.2 Optimal Curve: Average Number of Multiplications vs Cost in Number of ROM Words; VIVD Case; One Joint; Polyn. Deg. = 1,2 105

5.3 Optimal Curve: Average Number of Multiplications vs Cost in Number of ROM Words; VIVD Case; One Joint; Polyn. Deg. = 1,3 106

5.4 Optimal Curve: Average Number of Multiplications vs Cost in Number of ROM Words; VIVD Case; One Joint; Polyn. Deg. = 1,4 107

5.5 Optimal Curve: Average Number of Multiplications vs Cost in Number of ROM Words; VIVD Case; One Joint; Polyn. Deg. = 1,5 108

5.6 Optimal Curve: Average Number of Multiplications vs Cost in Number of ROM Words; VIVD Case; One Joint; Polyn. Deg. = 2,3 109

5.7 Optimal Curve: Average Number of Multiplications vs Cost in Number of ROM Words; VIVD Case; One Joint; Polyn. Deg. = 2,4 110

5.8 Optimal Curve: Average Number of Multiplications vs Cost in Number of ROM Words; VIVD Case; One Joint; Polyn. Deg. = 2,5 111

5.9 Optimal Curve: Average Number of Multiplications vs Cost in Number of ROM Words; VIVD Case; One Joint; Polyn. Deg. = 3,4 112

5.10 Optimal Curve: Average Number of Multiplications vs Cost in Number of ROM Words; VIVD Case; One Joint; Polyn. Deg. = 3,5 113

5.11 Optimal Curve: Average Number of Multiplications vs Cost in Number of ROM Words; VIVD Case; One Joint; Polyn. Deg. = 4,5 114

5.12 Ordering Relationship for Implementations Involving Two Different Polynomials (One Joint)..115

5.13 The Designer's Choice Among Two Possible Implementations 117

5.14 The Designer's Choice Among More Than Two Possible Implementations 119

5.15 Optimal Curve: Average Number of Multiplications vs Cost in Number of ROM Words; VIVD Case; Two Joints; Polyn. Deg. = 1,2,3 120

5.16 Optimal Curve: Average Number of Multiplications vs Cost in Number of ROM Words; VIVD Case; Two Joints; Polyn. Deg. = 1,2,4 121

5.17 Optimal Curve: Average Number of Multiplications vs Cost in Number of ROM Words; VIVD Case; Two Joints; Polyn. Deg. = 1,2,5 122

5.18 Optimal Curve: Average Number of Multiplications vs Cost in Number of ROM Words; VIVD Case; Two Joints; Polyn. Deg. = 1,3,4 123

5.19 Optimal Curve: Average Number of Multiplications vs Cost in Number of ROM Words; VIVD Case; Two Joints; Polyn. Deg. = 1,3,5 124

5.20 Optimal Curve: Average Number of Multiplications vs Cost in Number of ROM Words; VIVD Case; Two Joints; Polyn. Deg. = 1,4,5 125

5.21 Optimal Curve: Average Number of Multiplications vs Cost in Number of ROM Words; VIVD Case; Two Joints; Polyn. Deg. = 2,3,5 126

5.22 Optimal Curve: Average Number of Multiplications vs Cost in Number of ROM Words; VIVD Case; Two Joints; Polyn. Deg. = 2,4,5 127

5.23 Ordering Relationship for Implementations Involving Three Polynomials (Two Joints) 128

5.24 Global Ordering Relationship for Implementations with 2 or 3 Polynomials 129

5.25 Effect of Digitization on the Position of the Minimum. Two Joints. Polynomial Degrees 1,2,3 131

6.1 Sketch of the Arithmetic Unit for Function Evaluation 133

6.2 Type 1 Architecture: UIUD 141

6.3 VIUD Case; Variation of the h's Over the Interval . 144

6.4 Type 2 Architecture: VIUD 147

6.5 Barrel Shifter 148

6.6 Address Decoding Using a Barrel Shifter 150

6.7 Type 3 Architecture: UIVD 151

6.8 Type 4 Architecture: VIVD 153

7.1 The R-Method 158

7.2 Tradeoffs when $n \to \infty$ 159

8.1 Optimal Determination of the Joints for Several Functions Simultaneously 195

A.1 Probability Density Function of Floating Point Numbers between 0 and 1 221

LIST OF TABLES

Page

3.1 Number of ROM Words for Uniform and Non-Uniform Intervals 59

5.1 Conditions Used for Figures 5.2 through 5.22 ..102

1. INTRODUCTION

Since the publication of the article by Wallace [WAL64], giving an algorithm for performing division using multiplication as the basic operator, many researchers have considered methods for implementing high-speed division using the capabilities of a high speed multipliers [FER67], [STE72]. This research has also led to numerous papers on the generation of functions, such as 1/x and the trigonometric functions, by array-like function generators. The latter approach requires a different array for each function, resulting in an expensive process when many functions must be generated.

Considerable work has also been done in the area of digit-by-digit hardware function generation. De Lugish [DEL70], in a general approach to this problem, uses addition, subtraction, and shifting as the basic operations in his algorithms for implementing all arithmetic operations. An increase in speed for these types of sequential algorithms usually requires a redundant or high-radix number system. Numerous other papers have been published in this area [BAK73], [ERC75]. It is apparent, therefore, that the problem of sequential function generation has been thoroughly investigated.

This thesis deals with the following question:

> Is it possible to economically implement, in hardware, a numerical routine that performs faster than the traditional add-shift sequence of digit-by-digit routine?

Historically, sequential algorithms were devised to avoid using multiplication by replacing it with shifting, which was faster and easier to implement. It is feasible today, however, to implement high-speed parallel multipliers at reasonable cost. These multipliers, often of the Wallace-tree type, can be built from LSI blocks based on large Read Only Memories (ROM) [STE77].

This paper investigates an approach to function generation based on table look-up and on the minimum mean running time approximation of functions. The basic operator is multiplication and our argument is based on the existing high-speed hardware multipliers. Such multipliers can multiply two numbers in $O(\log n)$ gate delays. In order to consider algorithms having speed compatible with multiplication speeds, parallelism must be introduced at the hardware level. Most of the effort here is directed toward implementation of a function of one variable $f(x)$. The example $f(x) = 1/x$ is used for illustration.

The general organization of the thesis is as follows:

In Section 1, we briefly review the current parallel methods and give the rationale for the approach used here. Section 2 comprises a general discussion and a brief mathematical overview of the polynomial approximation methods. The applicability of piecewise polynomial approximation to hardware function generation is also considered. The choice of a criterion of "goodness of fit" is discussed and the nature of the distribution of the leading digits in floating-point numbers is introduced for use in the following sections.

In Section 3, the general architecture of high-speed function generators is presented and we distinguish four classes whose implementations in hardware increase in complexity. In this section the minimum mean running time approximation is introduced and it is shown that an optimal realization requires solving a non-trivial non-linear programming problem. In some instances this problem can be linearized or at least simplified.

In Section 4, we actually undertake to solve this problem using a penalty method. A FORTRAN program has been written to solve this, with $f(x) = 1/x$ as the example, for various combinations of polynomials in the range $[.5, 1]$. Some elementary considerations on the behavior of the minimum and a thorough study of the influence of various parameters lead to the results presented in

Section 5. Examples of optimum curves are presented for $f(x) = 1/x$ and polynomial degrees ranging from 1 to 5. The question of the choice of implementation given design constraints and the problem of address generation is discussed in Section 6. The four possible architectural types are described and some recommendations are made in view of the trends in ROM cost and size.

The complexity of the method is studied in detail in Section 7. Bounds for complexity in terms of memory requirements and bounds on the degrees of the polynomials are given. The method is also compared with a single-polynomial approximation approach.

In Section 8, the generation of other functions is briefly overviewed and the problem of the probability distribution of the exponents of floating-point numbers is discussed. Finally, suggestions for future research are offered.

1.1 Sequential vs. Parallel Function Generation Methods

Sequential algorithms for numerical computations have been extensively studied during the past five years. One of the original reasons for the study of digit-by-digit algorithms was the problem of high-speed hardware implementation of numerical routines for function evaluation. At the time the research was carried out, multiplication schemes were not implemented in parallel because of the high cost and the amount of required hardware $0(n^2)$. As a result, sequential algorithms were easily justified because the time required for evaluation, $0(n)$, was comparable to the time required for multiplication [HAB70], [BAU73]. Compared to numerical function evaluation routines using Taylor or Chebyshev series, which required several multiplications to attain a given precision, digit-by-digit algorithms presented a viable solution to the problem of high-speed function evaluation.

However, the continuing reduction in the size and cost of integrated circuit components and the advent of modular elements allowing a fast Wallace-tree reduction scheme [STE77], [SIN73] have made multiplication an operation economically implementable at speeds of $0(\log n)$ [DAD65]. The rigidity of the digit-by-digit algorithms makes them incompatible with the parallel evalutation

schemes which are applied to multiplication. Still, sequential algorithms find a broad range of applications in pocket calculators and special-purpose processors where speed is not a major concern.

The number of papers published on high-speed parallel algorithms for evaluation of commonly used functions that are implementable in hardware is surprisingly small [FER67], [CHI70], [FLY70].

It is interesting to quote [HAR68] concerning table look-up (in this case software) techniques:

> "Tabular data are an important element of machine evaluation of functions in two major cases: when the convenient source of quantitative data consists of a set of numerical values, and when the need for speed of evaluation outweighs considerations of storage and high precision."

Storage size has been a major factor in ruling out table look-up techniques in the literature. This is due to the fact that large scale ROMs and PLAs were not feasible as little as six years ago [now in 1978]. In addition, the only available basic operations were addition, subtraction and shifting. Today, LSI has made feasible long word length multipliers in very compact and efficient implementations [BRU73], [CHU74], [JOH73], [KIN71], [PEZ71] and multiplication with time of O(logn) should be fully included among the basic operations for numerical routines.

Next an overview of the parallel evaluation schemes considered in the literature is given.

1.2 Current Parallel Methods

The only major work in the area of parallel function evaluation was done by Tung Chin and Algirdas Avizienis in a paper entitled "Combinational Arithmetic Systems for the Approximation of Functions" [CHI70]. Their idea was to introduce a basic building block that allowed a modular implementation of Chebyshev polynomial or rational fraction evaluators. The set of polynomial coefficients was to be retrieved from a ROM and then used to evaluate the Chebyshev approximation of a given function f(x).

Some work has also focused on the area of high-speed evaluation of the function 1/x. These studies have been concerned with implementing fast division algorithms. There are several such algorithms in the literature dealing with the computation of the reciprocal, 1/x, of a normalized number x. Wallace developed an iterative division scheme making use of a fast multiplier [WAL64]. The IBM/360 Model 90 floating-point execution unit also makes use of a fast multiplier to perform division [AND67]. It uses an iterative process where, on the kth iteration, a factor R_k multiplies both numerator and denominator such that the resultant denominator converges quadratically toward one and the resulting numerator converges quadratically toward the desired quotient:

$$\frac{N}{D}\cdot\frac{R_o}{R_o}\cdot\frac{R_1}{R_1}\cdots\frac{R_n}{R_n} \longrightarrow \frac{N\cdot R_o\ R_1\ \cdots R_n}{1} \longrightarrow \text{quotient}$$

Ferrari [FER67] has also proposed a multiplicative scheme for computing the reciprocal of a number x. He bases his algorithm on interpolation of 1/x. Ling [LIN70] and Stefanelli [STE72] describe an inversion algorithm based on the Taylor series, making use of Read Only Memories and a set of dedicated, non-cascaded multipliers.

This latter technique is very fast because it is entirely parallel, but there are some inconveniences:

- The method uses dedicated multipliers to implement the division algorithm but it would seem that such an algorithm should be designed around an n x n multiplier that could be used for the multiplication operation alone, as well as for the generation of other functions.
- The Taylor polynomial is known to produce very poor behavior with respect to error; it is possible to use other approximating polynomials which yield better error control.
- It will be seen later that it is expensive to use algorithms based on series approximations and ROM look-up to implement a totally parallel divider for a precision of more than 32 to 34 bits. The precision can be increased, however, by more

efficient use of the ROMs than that proposed by Stefanelli.

It seems then that the flexibility of ROMs and the possibilities introduced by Programmable Logic Arrays have not really been exploited in the current literature. It is interesting, however, to review the two main existing hardware implementations: The Stefanelli approach to a fast inverse function generator and the general approach proposed by Chin and Avizienis.

1.2.1 Stefanelli's Inverting Circuit

Stefanelli [STE72] computes the reciprocal 1/x of a binary number x. His design yields an inverter requiring 4 k ROMs and 12 x 12 multipliers. The inversion algorithm is based on the McLaurin series. x is written as

$$x=1.b_1b_2 \ldots b_rb_{r+1}\cdot b_n \quad \text{or}$$

$$x=x_1+2^{-r}x_2$$

One can expand 1/x as

$$\frac{1}{x}=\frac{1}{x_1+2^{-r}x_2}=\frac{1/x_1}{1+2^{-r}\left(\frac{x_2}{x_1}\right)}=\frac{1}{x_1}\left[1-2^{-r}\left(\frac{x_2}{x_1}\right)+2^{-2r}\left(\frac{x_2}{x_1}\right)^2-\ldots\right]$$

By taking the first order approximation, the inverse is

$$\frac{1}{x} = \frac{1}{x_1} - \frac{2^{-r}}{x_1^2} x_2$$

This can also be viewed as a polynomial of the variable x_2 approximating $1/x$ between the two points $x = 1.b_1b_2\ldots b_r$ and $x_2 = 1.b_1b_2\ldots b_r + 2^{-r}x_2$. (See Figure 1.1.)

Stefanelli's technique consists of storing $1/x_1$ and $1/x_1^2$ (the coefficient of the variable x_2) in ROM and performing the multiplication by $2^{-r}x_2$.

It is not a very efficient algorithm. His approximation is equivalent to a piecewise polynomial approximation, where the intervals have size 2^{-r} and the polynomial is a Taylor polynomial. It will be seen that for the same amount of memory one can certainly reduce the error by using a better polynomial such as a Lagrange or Chebyshev polynomial. He also uses a set of dedicated multipliers to perform the multiplication; this is unnecessary as the multiplier could be used to generate several functions and, as such, provide a common hardware structure for several algorithms. And finally, the precision of the approximation is not uniform in the interval $[\frac{1}{2}, 1]$.

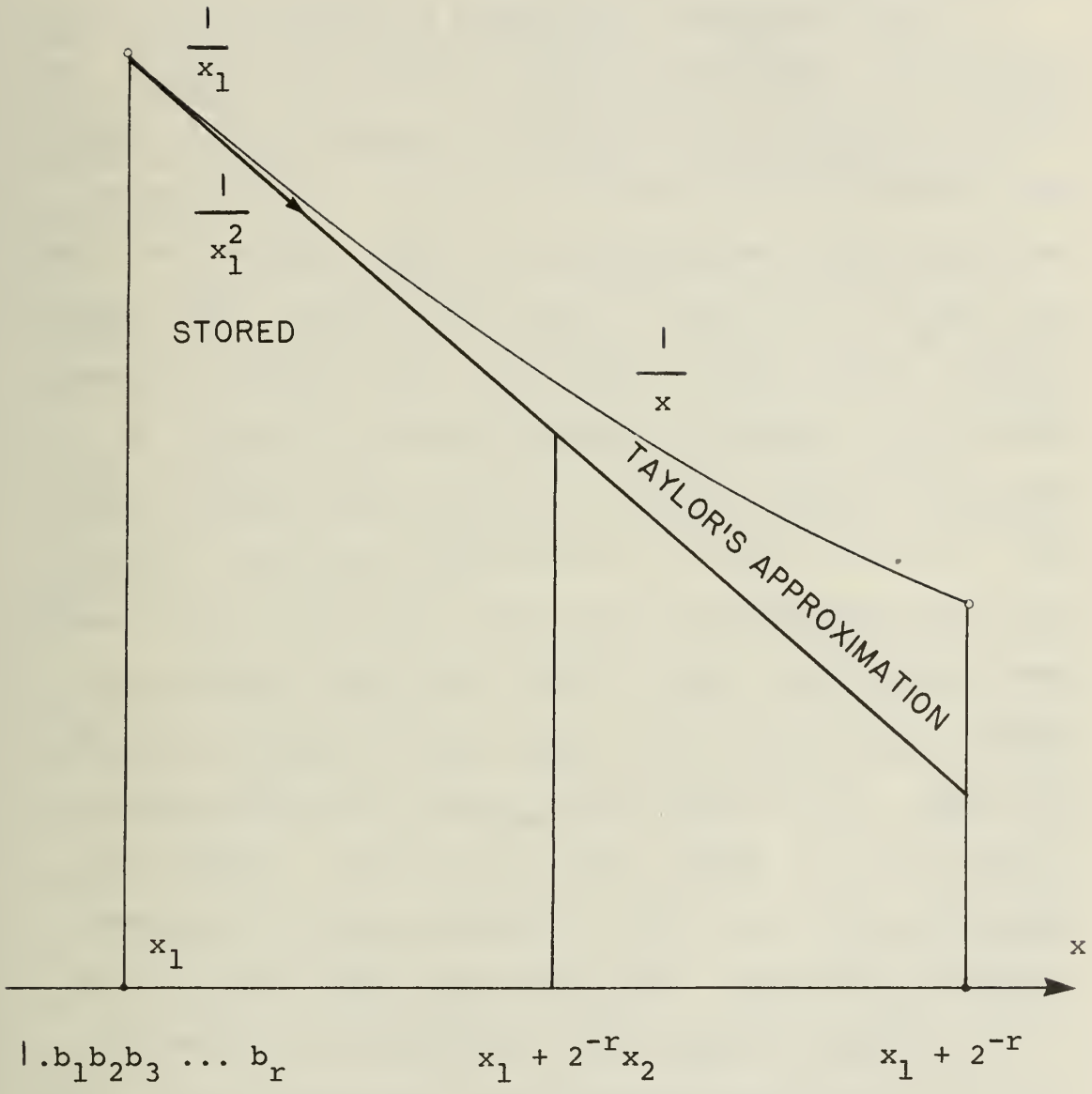


Figure 1.1 Stefanelli's Generation of 1/x

1.2.2 Chin's Method

Chin's approach constitutes an almost direct transposition of a subroutine library method into hardware. He defines the concepts of Arithmetic Building Blocks (ABBs) and Combinational Arithmetic nets (CAs). An ABB can be seen as a single package having an adder (called the "summer") which is an array of carry-save adders. This adder receives inputs from an "inputter" that converts a binary number into a minimally redundant radix 16 system. An "outputter" converts again into conventional binary form. Just to give an idea of the complexity involved, the number of I/O connections of the ABB is 126 plus the necessary power supply pins. For the whole ABB, the circuit complexity is the sum of 560 gates, 131 full adders, and 14 flip flops. The longest signal path in the ABB contains 19 gates, 20 full adders, and 1 flip flop. The ABB can generate a two digit (base 16) product plus the standard arithmetic functions. Chin uses many ABBs interconnected to form parallel arrays called combinational arithmetic (CA) nets or pipelined combinational arithmetic nets (PCA).

Each ABB can be considered to be a two-digit arithmetic unit that can perform the basic operations of addition, subtraction and multiplication. These building blocks can be connected so as to evaluate a polynomial in either a parallel fashion or a series fashion (pipelined

CA). Between these two extremes, the designer must make a trade-off between the cost of hardware and the speed of execution. Division is difficult in a CA net and this fact forced the author to choose polynomial approximation, which uses no division at all for the approximation of functions.

The main interest of the ABB is the modularity which allows pipelined techniques, but the cost of implementation is absolutely prohibitive in spite of the benefits of modularity. A 60-bit precision result would require between 2500 and 4000 of these ABBs and the speed would still be quite low.

In conclusion, it is apparent that although the modularity and flexibility of the design procedure using CA nets has certain advantages, the cost and actual hardware implementation would be extremely high and burdensome. The key point in Chin's approach is that a direct software to hardware implementation does not bring any improvement to the function generation problem. Clearly, some other methods must be investigated, using the possibilities of LSI, ROMs and PLAs, if high speed and simplicity of design are to be achieved.

1.3 Rationale for the Approach Used Here

While the design of mathematical function routines has already been thoroughly investigated from the numerical analysis point of view, the research on combinational arithmetic units for function generation has been limited to special array-type function units or division. Research in computer arithmetic has focused on digit-by-digit or sequential algorithms, which make use of the high-speed addition, subtraction and shifting capabilities of current computer technology [BAK73]. Higher radix number systems have been considered, with the basic motivation of increasing the speed of the sequential algorithms by the introduction of redundancy into the digit representation of the numbers [IRW77], [ERC75], [DEL70], [LAK76], [ROB76]. The impact of LSI on these algorithms has been studied from the point of view of modularity and research is currently focused on various types of Universal Arithmetic Blocks that can perform addition, subtraction, multiplication and division in a serial digit-by-digit fashion.

It seems, however, that insufficient attention has been given to the ever increasing capability of LSI modularity and speed and its potential application to combinational arithmetic units. Large and fast memories are now a part of today's generation of large-scale

computers [TOR75]. High-speed LSI has allowed the design of large combinational multipliers in very compact form. If the time to multiply two n-bit numbers has been decreased from O(n) to a feasible O(logn), it seems reasonable to attempt to obtain, in the same amount of time, the commonly used functions. The possibility of easy implementation of interpolation, including piecewise and spline interpolation, now exists at the hardware level. The object of this thesis is to study the application of the flexibility and versatility of Read Only Memories, Programmable Logic Arrays and LSI in general, to high-speed, parallel arithmetic.

2. POLYNOMIAL APPROXIMATIONS FOR HARDWARE FUNCTION GENERATION

In this section, the basic concepts of piecewise polynomial interpolation are considered and the statistical properties of floating-point numbers are reviewed. Rather than cumbersome and highly general mathematical definitions, a notation is used which, if not always rigorous, makes the development of the arguments understandable to both logic designers and mathematicians. The problem is neither to find the most efficient approximating polynomial nor to find the most general form. This has already been investigated at length in the numerical analysis literature. Our area of investigation is the practicality of implementation: how can one make judicious use of ROMs, PLAs and large-size multipliers to produce efficient, practical realizations? How can one increase the speed of function evaluation by applying techniques well known to numerical analysts to the problem of hardare polynomial approximation while at the same time taking into consideration the nature of the numbers, in this case, floating-point numbers in a computer?

First, floating-point numbers are not uniformly distributed and one should make use of this fact in

evaluating a function using these numbers. Secondly, what is important is not the speed of evaluation of f(x) over a given interval at a given point x but rather the statistical speed, that is, the global speed, the one that the user sees. The expected speed of evaluation in terms of the number of multiplications is the important question to consider. Given a function f(x) to be approximated in [a,b], the steps taken to obtain an approximation are the following:

(1) reduce the interval of definition of the function to a more suitable interval like $[1/\beta, 1]$ for a base β number representation system. This step is always included in any function evaluation algorithm and is sometimes called "normalization."

(2) subdivide the basic interval into subintervals such that the function can be approximated by a low degree polynomial; one that is lower in degree than that of polynomials used in software function generation, usually of degree 10 to 22.

This subdivision can be done after:

a. choosing the type of polynomial approximant

b. choosing the criterion for "goodness of fit," a measure criterion.

In the next section, the general method of piecewise polynomial approximation is presented in the light

of its potential use for hardware implementation with ROMs or PLAs. The logarithmic distribution of mantissas of floating-point numbers is then reviewed and the notation used throughout this work is presented for clarity of exposition.

2.1 Choice of a Criterion for Goodness of Fit

We will use, implicitly, the fundamental theorem of function approximation. Given a continuous function $f(x)$ on $I:[-1, 1]$ and a (Riemann) integrable weight function $w(x)$ which is positive on I (except, possibly, at a finite set of points of I at which $w(x) = 0$), there exists a unique polynomial $P_\mu(x)$, of degree μ, such that

$$||f-P_\mu||_2 = \left[\int_{-1}^{+1}(f(x)-P_\mu(x))^2 w(x)dx\right]^{\frac{1}{2}} \leq ||f-Q_\mu||_2 \qquad (2.1)$$

for any polynomial Q_μ of degree μ, different from P_μ where $||\cdot||_2$ is a notation for the norm:

$$||f||_2 \triangleq \sqrt{\int_{-1}^{1} f^2(x)w(x)dx}$$

This P_μ is called the least-squares approximation to f with respect to the weight function w(x) [CHE66], [CHE74], [SZE59], [TOD63].

Inequality (21) above is a consequence of a more general theorem involving not only the particular norm $||\cdot||_2$ (sometimes called the Root Mean Square norm) but a more general norm function satisfying the usual properties of a distance in a linear space.

There are several existing criteria for "goodness of fit" from which one can choose. First, one must choose between absolute difference type norms and relative error type norms. Here is a list of a few well-known measure criteria in use:

Least first power norm

$$\int_{-1}^{+1} |f(x)-P_\mu(x)|\,w(x)\,dx < \varepsilon_0 \qquad (2.2)$$

$$\int_{-1}^{+1} \left|\frac{f(x)-P_\mu(x)}{f(x)}\right| w(x)\,dx < \rho_0 \qquad (2.3)$$

Least square norm

$$\left[\int_{-1}^{+1}(f(x)-P_{\mu}(x))^{2}w(x)\,dx\right]^{\frac{1}{2}}<\varepsilon_{0} \qquad (2.4)$$

$$\left[\int_{-1}^{+1}\left[\frac{f(x)-P_{\mu}(x)}{f(x)}\right]^{2}w(x)\,dx\right]^{\frac{1}{2}}<\rho_{0} \qquad (2.5)$$

Uniform norm

max $|f|$

$$\max_{-1\leq x\leq 1}|f(x)-P_{\mu}(x)|<\varepsilon_{0} \qquad (2.6)$$

$$\max_{-1\leq x\leq 1}\left|\frac{f(x)-P_{\mu}(x)}{f(x)}\right|<\rho_{0} \qquad (2.7)$$

Additionally, the following two tests are sometimes used. (They are not norms but give additional information about the "quality" of the approximation.)

Bias absolute error= $\int_{-1}^{+1}(f(x)-P_{\mu}(x))w(x)\,dx$

$$\text{Bias relative error} = \int_{-1}^{+1} \left[\frac{f(x) - P_\mu(x)}{f(x)} \right] w(x)$$

Our choice is the RMS norm which is widely used and is well fitted to mathematical manipulations. We will use the notation:

$$C(f, P_\mu, x_1, x_2) = \sqrt{\int_{x_1}^{x_2} (f(x) - P_\mu(x))^2 w(x)\, dx}$$

where C stands for "Criterion."

The criterion for goodness of fit depends on:

$f(x)$: the function to be approximated,

$P_\mu(x)$: the polynomial approximant of degree μ, and

$w(x)$: the weighting function.

In general, we want to obtain a polynomial $P_\mu(x)$ satisfying

$$C(f, P_\mu, x_1, x_2) \leq \varepsilon_o$$

where ε_o is a predefined "error." We will use $\varepsilon_o = 2^{-n}$ for n-bit precision.

2.2 Colloquation Points

Fig. 2.1 illustrates the explanation that follows.

If the values of the function f(x) are known for a finite set of values of x in a given interval, then a polynomial which takes on the same values at these x abscissas is a simple analytic approximation to f(x) throughout the interval. This type of approximating technique is called polynomial interpolation. Its effectiveness depends on the choice of the nodes (or colloquation points). How to choose the μ points at which to sample any function in order to make the error as small as possible is a problem whose solution is still unknown and is not discussed here. The polynomial, of degree not exceeding μ, which agrees with f(x) at nodes x_1, x_2, x_3 ..., x_μ, is called the Lagrange interpolating polynomial. It is easily obtained if the nodes are known a priori. It is, in fact, much easier to find the interpolating polynomial, given the nodes, than to find <u>the</u> polynomial (and nodes) given one of the previous error criteria.

The error term for the Lagrange interpolating polynomial has been shown to be of the form

$$\pi_\mu(x) = f(x) - P_\mu(x) = \prod_{k=0}^{\mu} (x - x_k) \frac{f^{(\mu+1)}(\xi)}{(\mu+1)!} \tag{2.8}$$

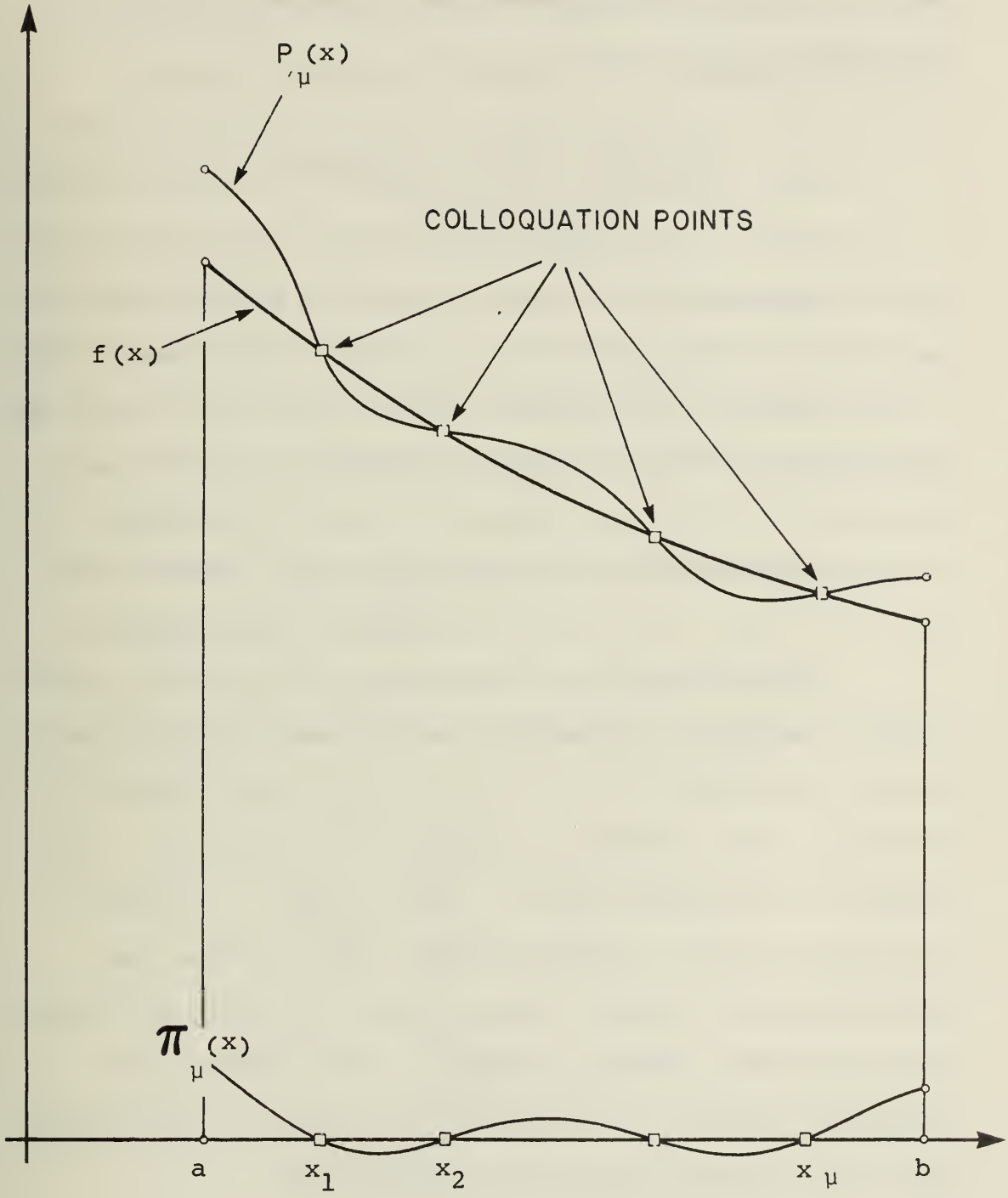


Figure 2.1 Colloquation Points

where ξ is located in the smallest interval containing the points $x_0, x_1, x_2, \ldots, x_\mu, x$.

2.3 Quasi-Optimal Polynomial Approximation

Equation (2.8) cannot be used to calculate the exact value of the error $f - P_\mu$, since ξ as a function of x is, in general, not known. Because there is nothing one can do about $f^{(\mu+1)}(\xi)$, the only effective way of minimizing the error is to minimize the maximum magnitude of the $(\mu+1)$st degree polynomial $\prod_{k=0}^{\mu}(x-x_k)$. The derivative $f^{(\mu+1)}(\xi)$ is treated as though it were constant.

Then the choice of the nodes $x_0, x_1, \ldots, x_\mu$ can be made, using the criterion for goodness of fit defined earlier, by taking $\pi(x) = B \prod_{k=0}^{\mu}(x-x_k)$ where B is a constant. This dictates the position of the nodes. For example, it is known that if $w(x)=\frac{1}{\sqrt{1-x^2}}$ and the criterion is (2.4), then the nodes that minimize the error criterion are the roots of the $(\mu+1)$st degree Chebyshev polynomial [RIV74], [LUT65]. This technique is sometimes used for its simplicity and ease of computation and it is the one which will be used here.

2.4 Piecewise Polynomial Approximation

Usually, when the function f is approximated over a large interval [a,b] by a single polynomial for the entire interval, the degree of the polynomial must be high if reasonable precision is required. In addition, there may also be some undesirable oscillatory properties. Separate approximations over subintervals smaller than [a,b] can improve the approximation to the function by:

a. requiring fewer terms of the polynomial approximant

b. adding flexibility to the form of approximants for each subinterval.

If the interval [a,b] is divided into subintervals I_1, I_2, ..., I_S, and if f(x) is approximated by a function F(x) of the form

$$F(x) = \begin{cases} P_{\mu_1}^{(1)}(x) & x \in I_1 \\ P_{\mu_2}^{(2)}(x) & x \in I_2 \\ \vdots & \\ P_{\mu_i}^{(i)}(x) & x \in I_i \\ \vdots & \\ P_{\mu_S}^{(S)}(x) & x \in I_S \end{cases}$$

where each $P_{\mu_i}^{(i)}$ is a polynomial of degree μ_i, then F is called an S-fold segmented approximation to f. $P_{\mu_i}^{(i)}(x)$ may be taken to be the best approximation to f on I_i. Then the deviation $||f-F||$ (where $||\cdot||$ is one of the norms) depends only on the subinterval breakpoints which define the subintervals I_i.

When the routines for approximating f(x) are implemented in software, two major drawbacks arise:

a. If the degree of the polynomial in each subinterval must be low (for fast computation), segmented approximation will frequently require many words of memory for storing the coefficients of the polynomials.

b. Given the argument x of f(x) is is not obvious how to determine the interval in which x lies. This is the problem of address computation, that is, location of the coefficients of the appropriate polynomial.

However, when the routine is implemented in hardware, one can make use of LSI to solve both a and b. Storing a thousand numbers in ROM is neither impossible nor impractical; 16K and larger Read Only Memories are common. 16K words of ROM easily fit on a small circuit card but this number of words has traditionally been considered to be a large amount of Read Only Memory. Four thousand 24-bit

coefficients easily fit side by side within the width of this page. As a consequence, it is not unrealistic to consider the possibility of hardware implementation for high-speed approximation of functions using piecewise low-degree polynomials.

The questions to be considered are:

- the choice of the subintervals which will maximize the speed and minimize the amount of ROM.
- the trade-offs between the speed and the memory requirements.
- the best architecture for efficient function generation.

Next, let us describe briefly a property of the distribution of mantissas of floating-point numbers which we will make use of in later sections.

2.5 The Logarithmic Distribution of Leading Digits in Floating-Point Numbers

A number of papers have been written discussing the distribution of the most significant digits in tables of physical constants. The logarithmic law of distribution is derived using various approaches. Pinkham and Raimi derive the logarithmic law by imposing certain assumptions on a statistical model. Benford and Flehinger [FLE66] find the law by analyzing the distribution of the leading

digits of the set of integers. Hamming [HAM70] gives a very clear account of the statistical properties of the mantissas of floating-point numbers and shows how the arithmetic operations of a computer transform various distributions toward the limiting reciprocal distribution

$$r(x) = \frac{1}{x \ln \beta} \qquad \left(\frac{1}{\beta} \leq x < 1 \right)$$

where β is the base number of the system. He also shows that for both multiplication and division, if one of the factors comes from the reciprocal distribution, the result also belongs to the reciprocal distribution independent of the distribution of the other factor. In other words, the reciprocal distribution persists under multiplication and division and cannot be changed by choosing any of the other factors. Some examples are given, such as the effect on the representation error of numbers in base 2 and base 16. Several other papers dealing with the study of roundoff errors [KAN73], [TSA74] include the logarithmic law as a weighting function to various measure functions.

This logarithmic distribution will be used in the remainder of this work. As will be seen, the position of the joints over the range $[1/\beta, 1]$ is influenced when taking this law into account.

2.6 Notation Used

The notation used in the remainder of the discussion is explained below and illustrated in Figure 2.2.

- The breakpoint subintervals are denoted by x_i where i may range from 1 to S. (This breakpoint index is often implicit in later sections.)
- The polynomial approximant for the ith interval $[x_i, x_{i+1}]$ is denoted $P_{\mu_i}^{(i)}(x)$ and is of degree μ_i.

It will be convenient later to divide the overall interval of interest into partitions within which the degree of the approximating polynomials are the same. These partitions are denoted by $[X_j, X_{j+1}]$ where the X_j are called joints. There are s such intervals. Since μ_i is unchanged in $[X_j, X_{j+1}]$, it is convenient to denote it by m_j. This leads to the notation $P_{m_j}^{(i)}(x)$ with i ranging from 1 to S and j ranging from 1 to s.

To summarize:

- The partition joints are denoted by X_j (capital X).
- The polynomial approximants are $P_{m_j}^{(i)}(x)$ between joints X_j and X_{j+1}, i.e., within a given partition.
- m_j is the degree of this polynomial in $[X_j, X_{j+1}]$.
- There are s partitions and s+1 joints.
- It is understood that there is a different polynomial $P_{m_j}^{(i)}(x)$ for each breakpoint interval $[x_i,$

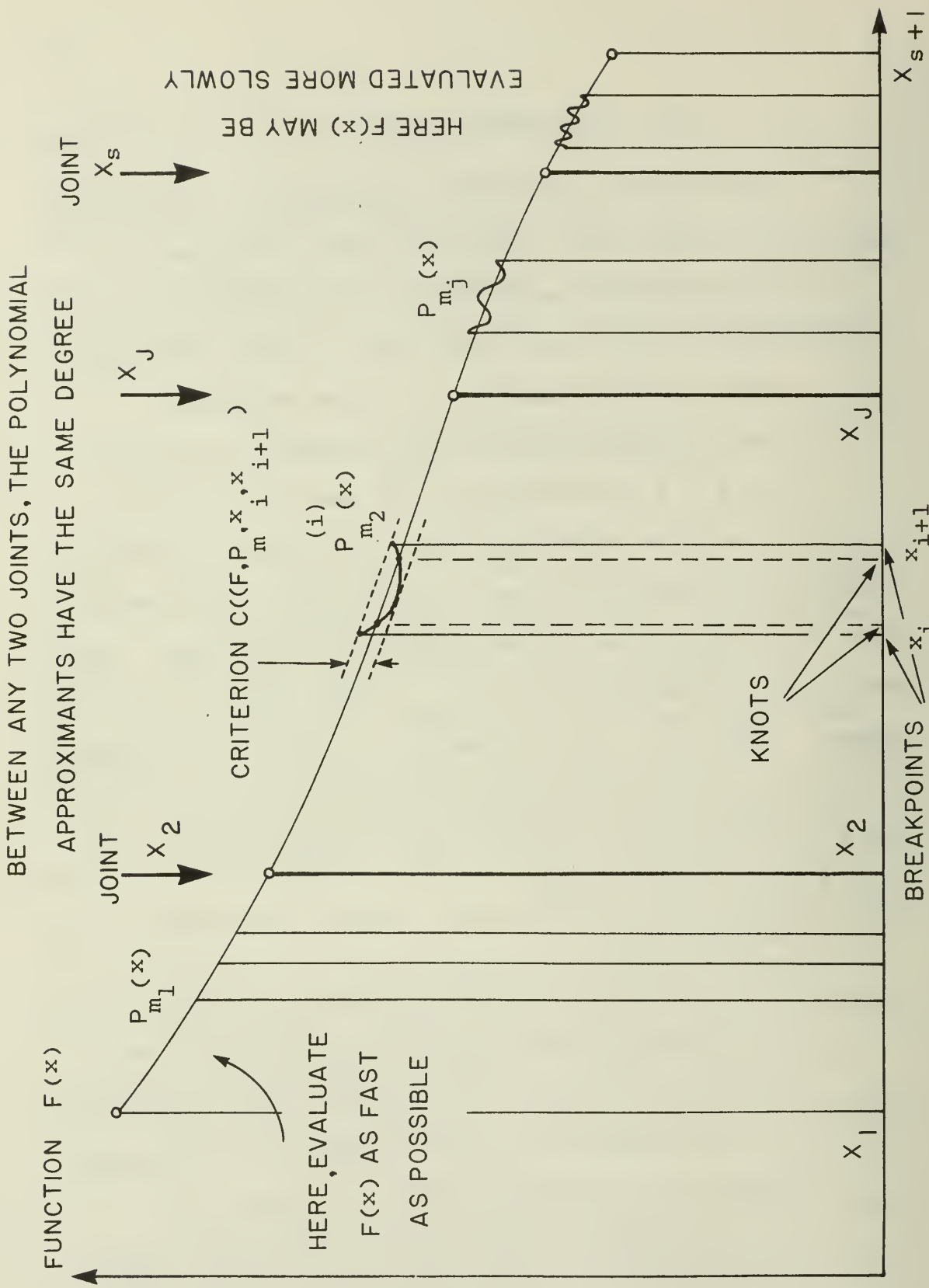


Figure 2.2 Notation Used Throughout the Discussion

$x_{i+1}]$. The index i is implicit when we refer to $P_{m_j}(x)$.

- $C(f, P_m, x_i, x_{i+1})$ is the "goodness of fit" criterion as defined earlier.
- β is the base of the number representation.

3. ARCHITECTURE FOR HIGH-SPEED FUNCTION GENERATION: FOUR CLASSES

In this section, we will introduce one of the main ideas in this discussion, the concept of the minimum mean running time approximation. This will yield a non-linear programming problem that we will attempt to solve in the next sections. Before setting the general non-linear programming problem, it will be necessary to have some way of computing the number of breakpoints for a given criterion $C(f, P_m, x_1, x_2)$ and ε_o over a given range [a, b]. This is discussed in Section 3.3.2. First, four schemes for piecewise function approximation are presented. Given an argument x represented as a normalized floating-point number, and assuming that the range [a, b] has been reduced to the interval $[1/\beta, 1]$, one must answer the following questions:

a. How should one choose the breakpoint locations for the piecewise polynomial approximation?

b. How should one choose the degrees for each one of these piecewise polynomials?

The four choices are described below.

3.1 Uniform Interval Length Using Uniform Degree Polynomials (UIUD)

The first reasonable choice is a uniform mesh of breakpoints with a fixed polynomial degree over the whole interval range $[1/\beta, 1]$. The width of each interval will be called h and the breakpoints, x_i.

$$h = x_{i+1} - x_i$$

It is natural to choose an interval length such that

$$h = 2^{-\ell}$$

where ℓ will depend on a trade-off between speed and memory size. This choice is natural because the value of x_i such that

$$x_i \leq x \leq x_i + 2^{-\ell}$$

is extracted from the argument x itself simply by taking the ℓ most significant bits of the normalized argument x (Fig. 3.1). Consequently, the address of the coefficients of the ith polynomial approximating f(x) between x_i and $x_i + 2^{-\ell}$ is represented by those ℓ most significant bits. Moreover, if the base β is 2 then, knowing that the first bit is a 1, only $(\ell - 1)$ bits are necessary for generating the address of the ith polynomial.

When the intervals have a uniform length = $2^{-\ell}$, the position the ith interval can be extracted from x directly.

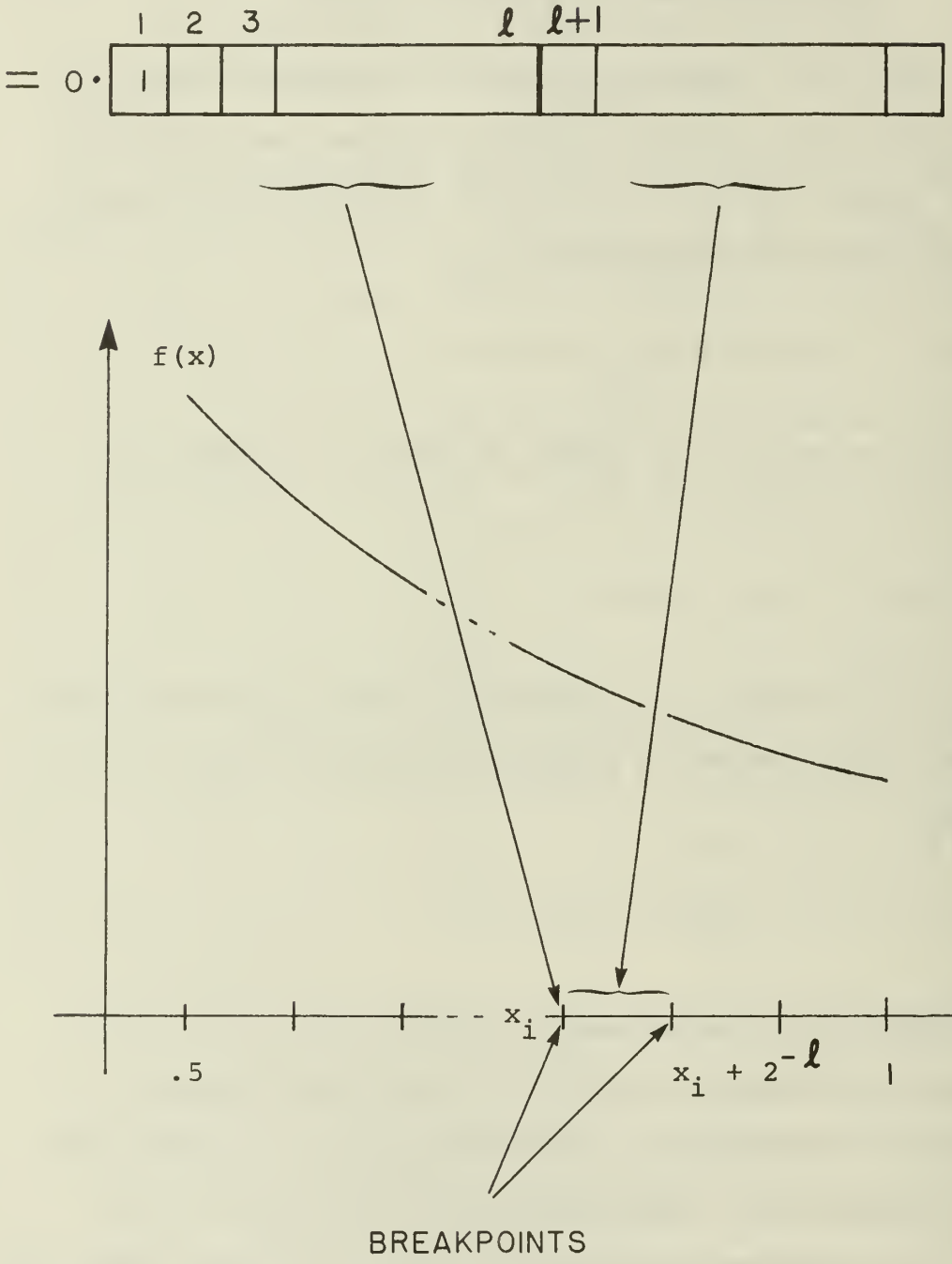


Figure 3.1 Uniform Interval Length Case

The design procedure for this case, where the interval size is uniform as well as the degree of the polynomial, is flow-charted in Fig. 3.2. This case will be referred to as the UIUD (Uniform Interval, Uniform Degree) case.

3.2 Variable Interval Length Using Uniform Degree Polynomials (VIUD)

It should be realized that the value of $C(f, P_m, x_i, x_{i+1})$ in the previous scheme will generally not remain constant for all $x_i \in [1/\beta, 1]$. The only thing that the above procedure assures is the choice of h for the worst case (which depends on the function f). As an example, the function 1/x for x between .5 and 1 with $h = 2^{-10}$ and a Lagrange polynomial of degree 1 in each interval, shows (Fig. 3.3) that the criterion value changes by a factor of 10 over the interval. For a polynomial of higher degree the change would be even greater.

Consequently, it seems that some ROM can be saved by having variable length intervals h_i. Given x_i, the following non-trivial equation must be solved recursively for h_i and $P_m(x, h_i)$:

$$\int_{x_i}^{x_i+h_i} (f(x)-P_m(x,h_i))^2 w(x)\,dx=\varepsilon_0^2 \qquad (3.1)$$

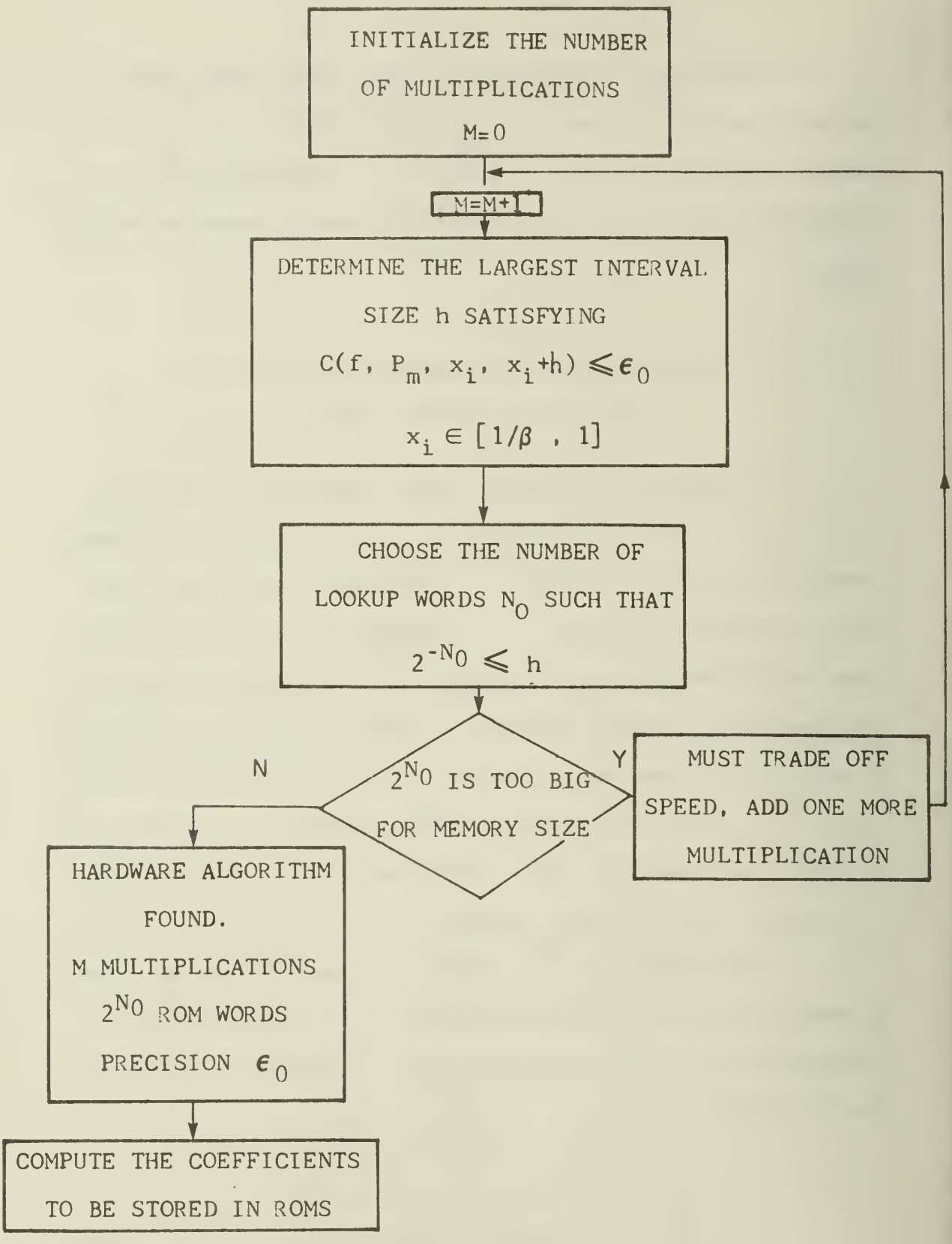


Figure 3.2 UIUD Design Procedure

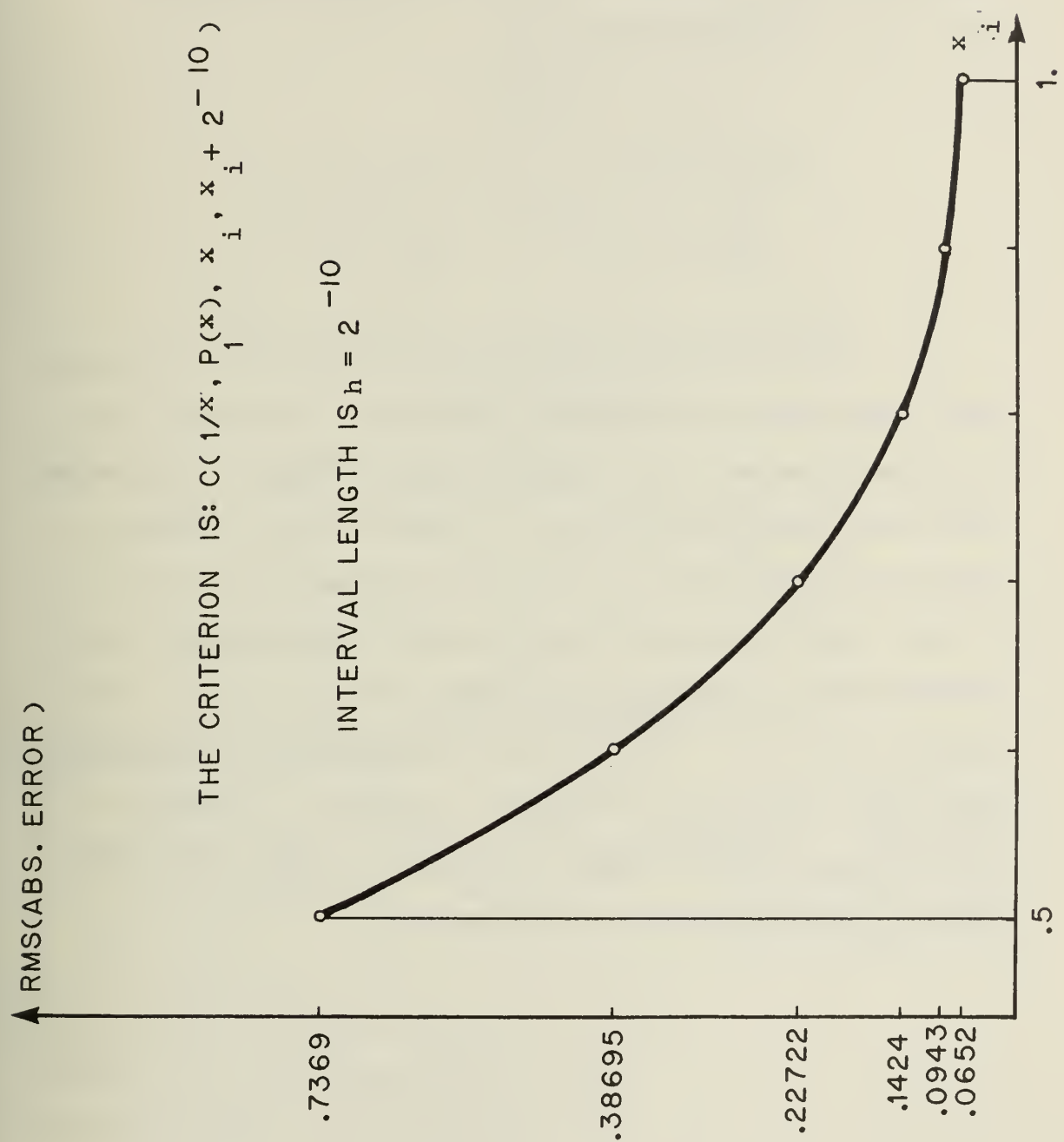


Figure 3.3 Illustration of C(f, P_m, x_i, x_i + h) as a Function of x_i for f = $\frac{1}{x}$

where the notation $P_m(x, h_i)$ is used to indicate that the choice of $P_m(x)$ depends on the breakpoint spacing h_i, with the initial condition

$$\int_{\frac{1}{\beta}}^{\frac{1}{\beta}+h} (f(x)-P_m(x,h))^2 w(x)\,dx=\varepsilon_0^2$$

The problem created by this procedure at the hardware level is in the address generation. Because of the variable intervals, the address can no longer be extracted from the argument x directly. A ROM or PLA is needed in order to provide the correspondence between the ℓ (or ℓ - 1) most significant bits of x and the ROM address of the coefficients. The design procedure for this case is outlined in Fig. 3.4. Table 3.2 in Section 3.3.2 (page 59) shows that for f = 1/x and $P_m(x)$ of degree 1 with $h = 2^{-10}$, the savings in the number of ROM words is 40%. Thus the address decoding memory is worthwhile.

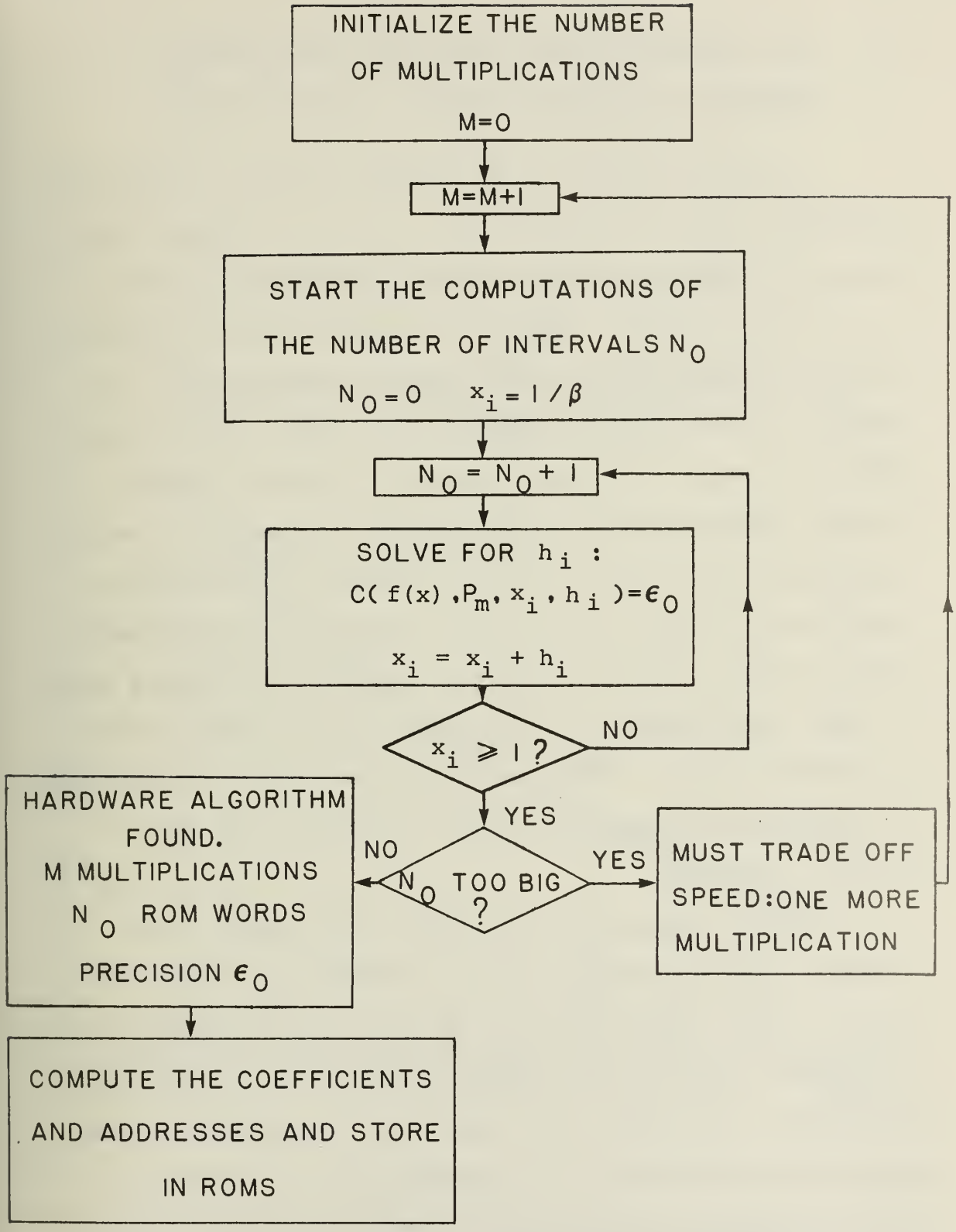


Figure 3.4 VIUD Design Procedure

3.3 Variable Degree Polynomials with either Uniform or Variable Intervals (UIVD and VIVD)

So far (see Fig. 3.5), we have chosen a single polynomial degree over $[1/\beta, 1]$. If this polynomial has degree m, we know that the speed of function evaluation is m multiplications. However, it is not always possible to choose a polynomial of low degree -- degree one, for example -- if we wish to evaluate f(x) with, say, 64 bits precision, due to the large number of ROM words needed. The only solution then is to increase the number of multiplications to 2, 3 or 4 until the number of intervals is less than or equal to that which results in an affordable amount of ROM. This is somewhat inflexible and would seem to limit the choices for a designer to an integral number of multiplications. This is because we have not considered the statistical speed, the speed with which the user is actually concerned.

If one computes f(x) 10,000 times, what is the "average" speed? Perhaps we can implement the function generator over one-half of the interval with polynomials of degree 1 and over the remaining half with polynomials of degree 2 (see Fig. 3.5). Assuming a uniform distribution of the arguments x, on the average the number of multiplications necessary to evaluate our function is 1.5.

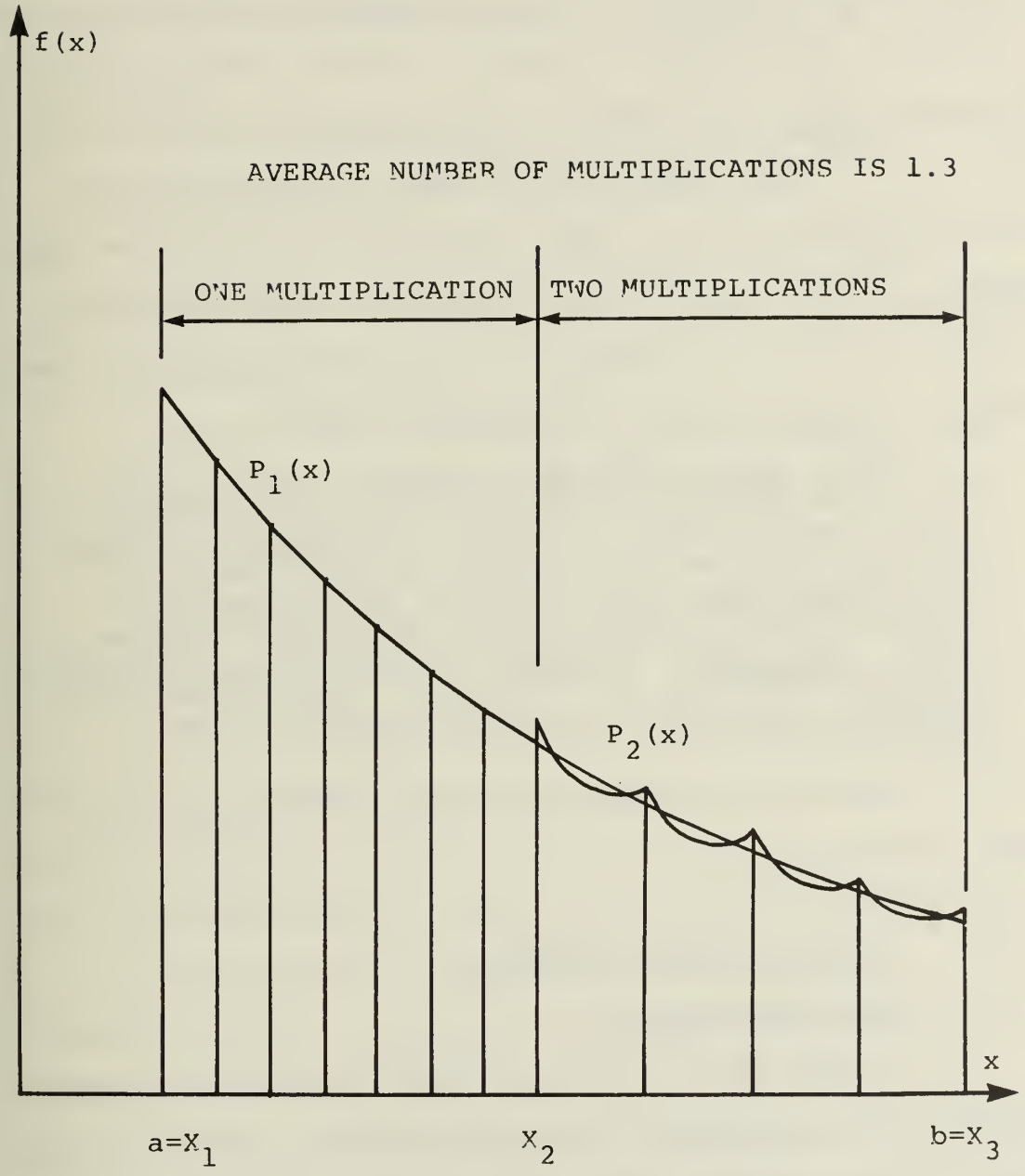


Figure 3.5 Variable Degree Polynomials with either Uniform or Variable Intervals (UIVD and VIVD).

Sometimes, x falls in the 2 multiplication range while at other times only 1 multiplication will be necessary. Moreover, when we consider the non-uniform distribution of the arguments, with smaller values of x more likely, x will certainly fall more often in the first half of the range so that, in fact, the average number of multiplications may be closer to 1.3. Thus, the distribution of the arguments x of the function to be evaluated will affect the choice of the partitions and, quoting Hamming in his famous paper on the distribution of numbers [HAM70]:

> "It is easy to see as a general rule that when we try to optimize a library routine for minimum mean running time (as against the Chebyshev minimax run time) we need to consider the distribution of the input data. Hence floating point numerical routines need to consider the reciprocal distribution: the square root, log, exponential, and sine are all examples. In the case of the exponential and sine, some study of the exponents is also necessary."

The choice of the partitions is the subject of the next section.

3.3.1 The Minimum Mean Running Time Approximation

We have seen earlier (Section 2.5) that not all values of the arguments x are equiprobable. This means that, on the average, more time is spent evaluating f near $x = 1/\beta$ than near $x = 1$. Why not, then, use a low degree polynomial where the probability of finding x is

low? In other words, the fact that values of the argument x around 1 are less probable than around $1/\beta$ can be utilized by computing the function using, for example, the following strategy:

- fast (1 multiplication, $P_m(x)$ is of degree 1) where the number x occurs often (around $1/\beta$).
- less fast (2 multiplications, $P_m(x)$ of degree 2, and larger intervals) where x occurs less frequently (somewhere between $1/\beta$ and 1).
- slowly (3 multiplications, $P_m(x)$ of degree 3 and even larger intervals) where x is not found too often (around 1).

One would then consider the average number of multiplications over the entire interval. The problem consists of minimizing the mean running time depending on the positions of the "joints" (the X_j's in Fig. 2.2). As an example, assume that there are four joints, X_1, X_2, X_3, and X_4, and that the degrees of the polynomials in the 3 intervals are 1, 2, 3. The number of breakpoint intervals in $[X_j, X_{j+1}]$ is denoted as $N(X_j, X_{j+1})$ and the number of multiplications necessary to evaluate f at x, $x \varepsilon [X_j, X_{j+1}]$, is m_j. The number of multiplications needed at any point x is denoted $M(x)$.

The first condition that must be satisfied is to minimize the average number of multiplications M_{av} over the whole interval.

$$\min\left[M_{av} = \int_{X_1}^{X_4} \frac{1}{x} M(x)\,dx \right] =$$

$$\min\left[1 \ln\left(\frac{X_2}{X_1}\right) + 2 \ln\left(\frac{X_3}{X_2}\right) + 3 \ln\left(\frac{X_4}{X_3}\right) \right] \qquad (3.2)$$

The second constraint is to keep the number of ROM words under some maximum, R_o. This is stated as:

$$2N(X_1, X_2) + 3N(X_2, X_3) + 4N(X_3, X_4) \leq R_o \qquad (3.3)$$

(3.2) can be rewritten as:

$$\min\left[\ln\left(\frac{X_2}{X_1}\right)^1 \left(\frac{X_3}{X_2}\right)^2 \left(\frac{X_4}{X_3}\right)^3 \right]$$

which becomes:

$$\max\ (X_1 \cdot X_2 \cdot X_3 \cdot X_4) \qquad (3.4)$$

Condition (3.3) is non-linear for the case of non-uniform intervals and depends on the precision desired and

the behavior of f(x). However, if we consider the case of uniform interval breakpoints, condition (3.3) becomes linear. (3.3) and (3.4) form a Non Linear Programming problem whose solution is the vector of joint positions $X_2, \ldots, X_s$. (x_1 and x_{s+1} are the fixed interval boundaries.)

The solution of this problem is the optimum position of the X_j's for the lowest average number of multiplications with a constraint on the number of ROM words available.

This type of approximation will be referred to as the minimum mean running time or MMRT approximation. It is quite different from the traditional Chebyshev minimum-maximum (minimax) running time approximation used in software numerical routines.

We must now answer two questions:

1. How can we compute the number of breakpoints, $N(X_j, X_{j+1})$, between two joints?
2. How can we solve the non-linear programming problem stated above?

The next section is devoted to the search for an algorithm for finding the number of breakpoints in the interval $[X_j, X_{j+1}]$. Knowing the degree of the polynomial approximating f(x) between X_j and X_{j+1}, the number of ROM words needed for the piecewise approximation of f(x) for x between the two joints can be derived. If we assume that

the polynomial is of degree m_j and that it is neither even nor odd and that no coefficient is null, then the number of ROM words is $N(X_j, X_{j+1})\ (m_j+1)$ since the number of coefficients required is one greater than the degree of the polynomial in that interval. The number of multiplications between X_j and X_{j+1} is m_j,* the degree of the polynomial. We assume that the criteria for "goodness of fit" is the RMS criterion defined earlier.

3.3.2 Algorithm for Finding the Number of ROM Words

Choosing the RMS norm for $C(f, P_m, x_i, x_{i+1})$ requires the solution of the following equation for h:**

$$\int_{x_i}^{x_i+h} (f(x)-P_m(x_i,h))^2 w(x)=\varepsilon_o^2 \qquad (3.5)$$

---

*If a parallel polynomial evaluation scheme is used, then the number of multiplications is bounded by $2\log(m_j)$ and condition (3.2) changes due to the presence of $2\log m_j$ instead of m_j. This will affect the position of the joints. However we assume that we evaluate $P_{m_j}(x)$ serially with one processor, using Horner's rule. The more general case is left for further study.

**It is understood that h = h(x). That is, each interval $[X_i, X_{i+1}]$ may have a different h. Also $P_m(x,h)$ is, in this case, $P_{m_j}(x,h_i)$. We are considering a breakpoint interval somewhere between two joints.

If instead of using the absolute error $|f - P_m|$, one decides to use the relative error, one would choose

$$w(x) = \frac{w_1(x)}{f(x)^2}$$

The weight function w(x) can penalize the error by weighting it more or less depending on the interval $[x_i, x_i+h]$.

Equation (3.5) is an equation in h. Notice that h appears in the polynomial P_m (degree m) because this polynomial approximates f in the interval $[x_i, x_i+h]$. Solving the equation for the adjacent interval requires knowledge of the value of h for the previous interval. We can therefore infer that the position of x_{i+1} is dependent on the position of x_i, x_{i-1}, ... It is desirable to find a general relationship between x_{i+1} and x_i:

$$x_{i+1} = G(x_i)$$

An approximate solution would give us an idea of the advantage of having variable interval lengths versus uniform interval lengths, in terms of ROM size. Equation (3.5) is not in an easily usable form. First recall the relation:*

$$f(x) - P_m(x_i, h) = \prod_{k=0}^{m} \frac{(x - x_k^i) f^{(m+1)}(\xi_i)}{(m+1)!}$$

*Again, $P_m(x,h) \equiv P_{m_j}^{(i)}(x, h_i)$ and we are considering the interval $[x_i, x_{i+1}]$.

where $f^{(m+1)}$ is the (m+1)st derivative of f and the x_k's are the roots of the Chebyshev polynomial of degree m between x_i and x_{i+1}. (We have limited ourselves to Chebyshev approximating polynomials.)

ξ_i is anywhere between x_0 and x_m (the first and last polynomial roots) but also depends on x.

We will now make the approximation

$$f^{(m+1)}(\xi_i) \simeq f^{(m+1)}(x_i)$$

Equation (3.5) can then be written:

$$\left[\frac{f^{(m+1)}(x_i)}{(m+1)!}\right]^2 \int_{x_i}^{x_i+h} \prod_{k=0}^{m} (x-x_k^i)^2 w(x)\,dx = \epsilon_0^2 \qquad (3.7)$$

If $f^{(m+1)}$ is monotonically increasing, equation (3.7) will give us a lower bound on the number of intervals. A monotonically decreasing equation (3.7) will yield an upper bound on the number of breakpoints. In the discussion that follows, we assume that the functions we are considering are "well-behaved" so that these assumptions apply.

A change of variable will further simplify equation (3.7). Let us call r the new variable which maps interval $[x_i, x_{i+h}]$ onto the interval [0,1]:

$$r(x) = r = \frac{x-x_i}{h} \text{ or } x = x_i + rh$$

Then (3.7) becomes:

$$h^{2m+3}\left[\frac{f^{(m+1)}(x_i)}{(m+1)!}\right]^2\int_0^1 \prod_{k=0}^{m}(r-\alpha_k)w(x_i+rh)dr = \epsilon_0^2 \quad (3.8)$$

where α_k's are the roots of the normalized Chebyshev polynomial of degree m. For example, if m = 2:

$$\alpha_1 = \frac{1+\frac{\sqrt{2}}{2}}{2} \quad \text{and} \quad \alpha = \frac{1-\frac{\sqrt{2}}{2}}{2}$$

and in general:

$$\alpha_k = \frac{1}{2}\left[\cos\left(1+2(k-1)\cdot\frac{\pi}{2m}\right)+1\right]$$

Notice that if $w(x) \triangleq 1$, equation (3.8) has the form:

$$h^{2m+3}[f^{(m+1)}(x_i)]^2 K_m = \epsilon_0^2 \quad (3.9)$$

where K_m is the integral:

$$\frac{1}{[m+1)!]^2}\int_0^1 \prod_{k=0}^{m}(r-\alpha_k)^2 dr \quad (3.10)$$

which does not depend on x_i or h but only on the type of polynomial approximant. If w(x) is not equal to 1, we can make, for h small enough, the approximation: $w(x_i + rh) \simeq w(x_i)$, and equation (3.8) can be expanded and will have terms in h^{2m+3}, h^{2m+4}, etc., but is more difficult to solve analytically. We will therefore limit this study to the first term of the equation. This leads to the final form:

$$h^{2m+3}\left[\frac{f^{(m+1)}(x_i)}{(m+1)!}\right]^2 w(x_i)\int_0^1 \prod_{k=0}^{m}(r-\alpha_k)^2 dr = \epsilon_0^2 \qquad (3.11)$$

or

$$h^{2m+3}[f^{(m+1)}(x_i)]^2 w(x_i) K_m = \epsilon_0^2 \qquad (3.12)$$

Solving for h:

$$x_{i+1}-x_i = h = \left[\frac{\epsilon_o^2}{K_m}\cdot\frac{1}{[f^{(m+1)}(x_i)]^2 w(x_i)}\right]^{\frac{1}{2m+3}}$$

or

$$x_{i+1}-x_i = \frac{k_m}{\left[[f^{(m+1)}(x_i)]^2 w(x_i)\right]^{\frac{1}{2m+3}}} \qquad (3.13)$$

where $k_m = \left(\frac{\epsilon_0^2}{K_m} \right)^{\frac{1}{2m+3}}$

This relation, of the form:

$$x_{i+1} - x_i = k_m \, g(x_i)$$

can be represented pictorially as shown in Fig. 3.6, 3.7, and 3.8 for three different types of functions g(x).

Starting from x_0 we obtain

$$x_1 = x_0 + k_m \, g(x_0)$$

$$x_2 = x_1 + k_m \, g(x_1)$$

$$\vdots$$

$$x_{i+1} = x_i + k_m \, g(x_i)$$

etc., until the end of the interval is reached.

If the number of breakpoints S is large enough, one can derive an expression for the ratio of the number of breakpoints in the non-uniform interval case to the number of breakpoints in the uniform interval case.

Relation (3.13) shows that if the magnitude of $f^{(m+1)}(x_i)$ increases with x_i, the smallest interval would be $[x_0, x_1]$. (See Fig. 3.6.) In the uniform interval case, all the intervals would have to be chosen equal to $[x_0, x_1]$ to guarantee $||f - P_m|| \leq \epsilon_o$.

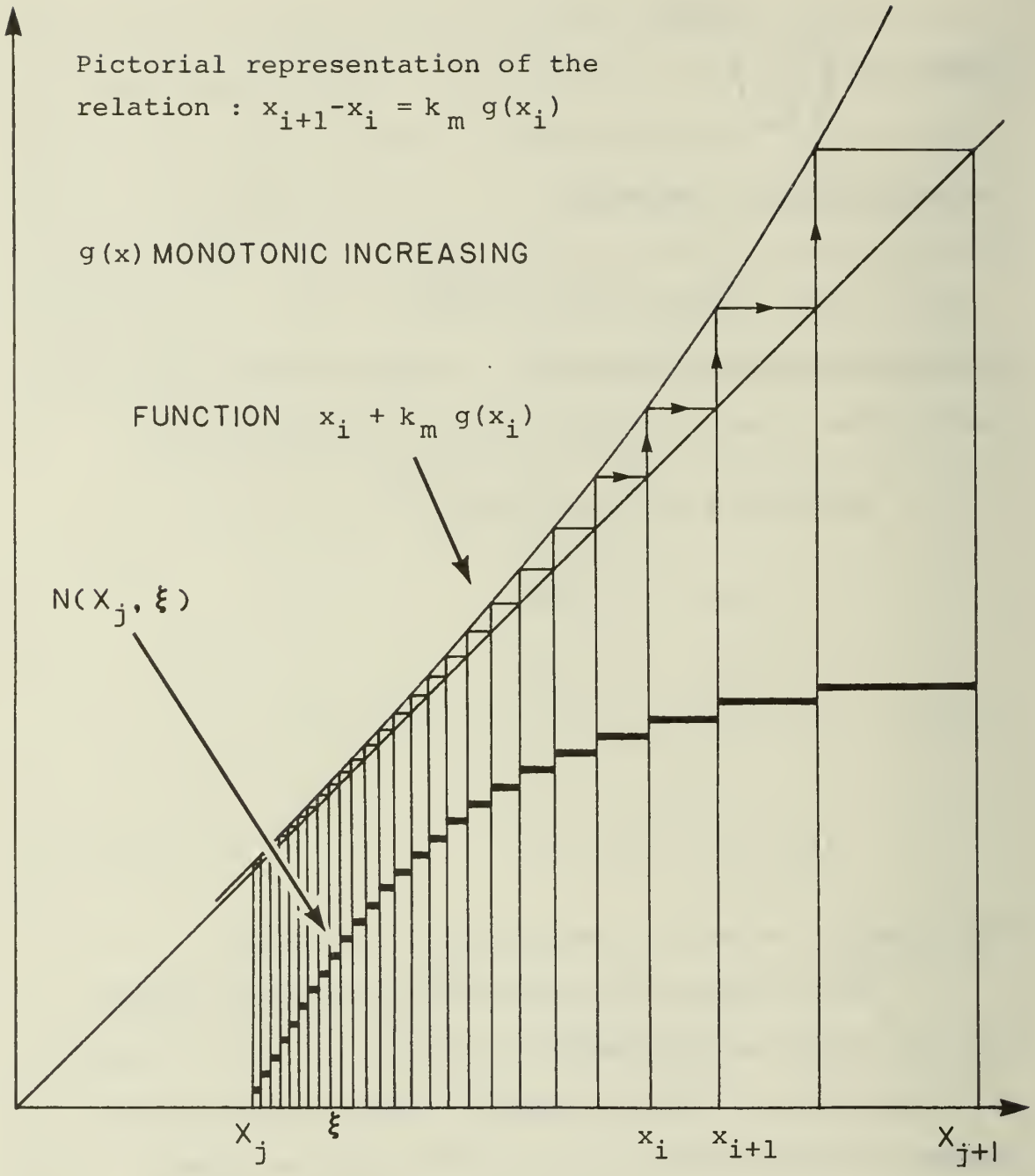


Figure 3.6 Number of Breakpoints in the Non-Uniform Case

If the magnitude of $f^{(m+1)}(x_i)$ decreases with x_i, the smallest interval would be the last one (right most) $[x_S, x_{S+1}] = [x_S, X_{j+1}]$ and its magnitude would be $k_m g(x_S) \simeq k_m g(x_{j+1})$. (See Fig. 3.7). We will thus assume that the smallest breakpoint interval is at either extremity of the partition $[X_j, X_{j+1}]$.

Let us assume then that the smallest interval is $[X_j, x_1] \equiv [x_0, x_1]$.

The number of breakpoints in $[x_0, x]$ will be given by the function $N(x_0, x)$.

From x_i to x_{i+1} $N(x_0, x)$ increases by one.

We then have:

$$\frac{N(x_0,x_{i+1})-N(x_0,x_i)}{x_{i+1}-x_i} = \frac{1}{x_{i+1}-x_i} = \frac{1}{k_m g(x_i)}$$

If (x_i, x_{i+1}) is small enough, this is close to

$$\left.\frac{dN}{dx}\right|_{x=x_i} = \frac{1}{k_m g(x_i)}$$

and therefore

$$N(x_0,x) = \frac{1}{k_m}\int_{x_0}^{x} \frac{du}{g(u)}$$

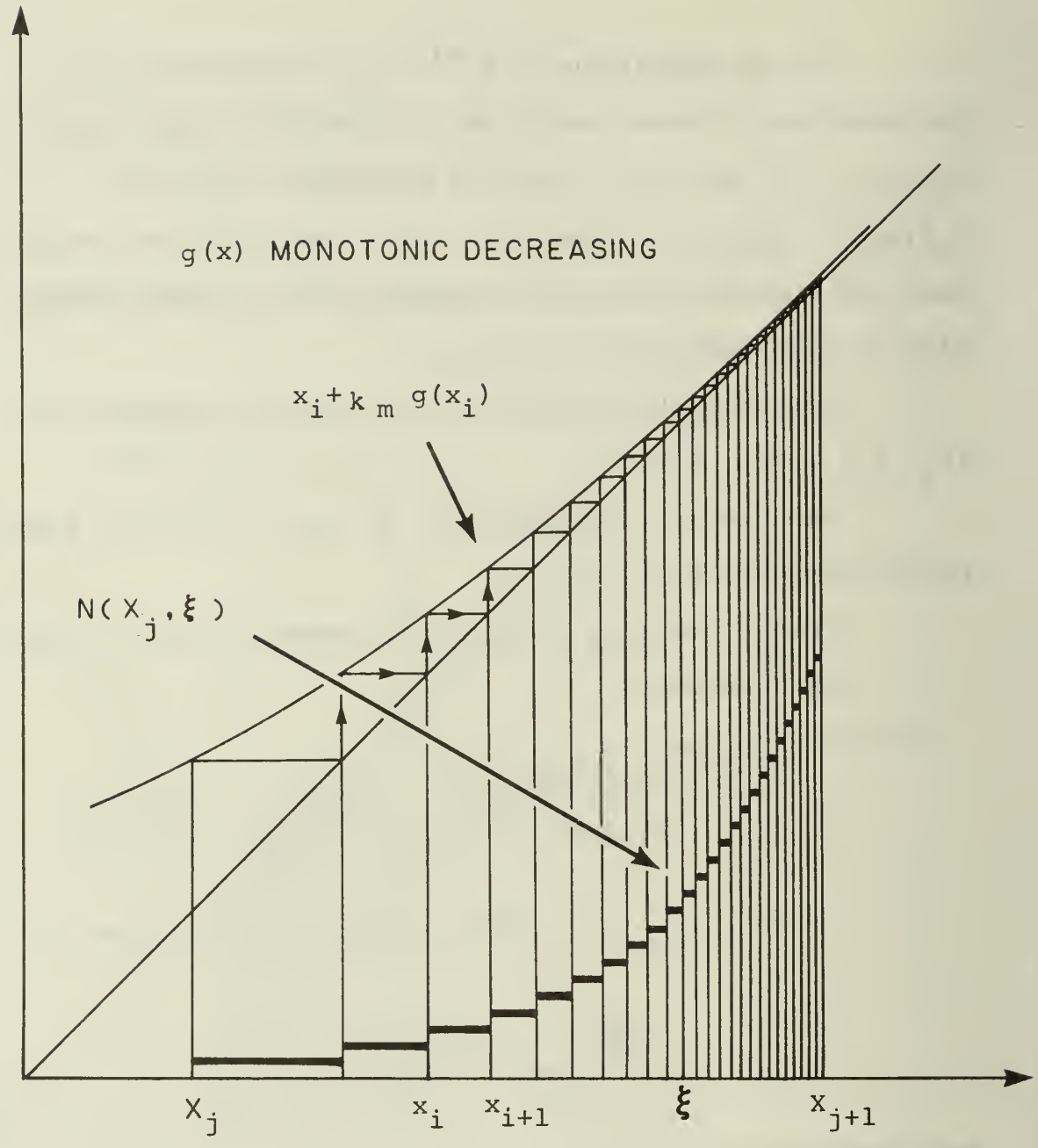


Figure 3.7 Number of Breakpoints in the Non-Uniform Case; Different Behavior of $N(x_o, x)$

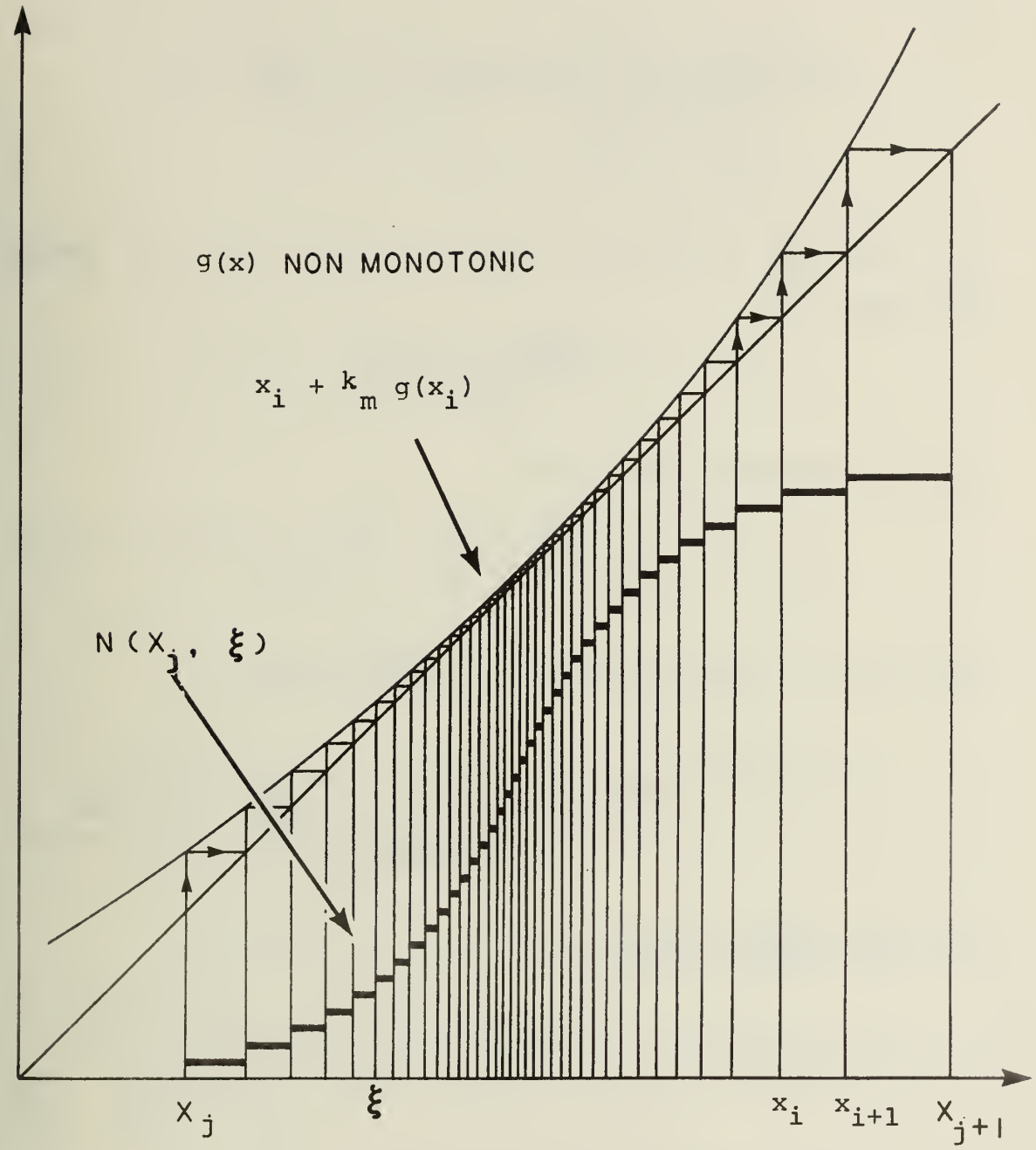


Figure 3.8 Number of Breakpoints in the Non-Uniform Case; Another Behavior of $N(x_o, x)$

$$N(x_0,x) = \frac{1}{k_m} \int_{x_o}^{x} \left[[f^{(m+1)}(u)]^2 w(u) \right]^{\frac{1}{2m+3}} du$$

The number of breakpoints in the interval $[x_o,x]$ for the uniform case is:

$$N_u(x_o, x) = \frac{x - x_o}{k_m g(x_o)}$$

or, in the non-uniform case

$$N(x_o, x) = \frac{1}{k_m} \int_{x_o}^{x} \frac{d_u}{g(u)}$$

The ratio of the non-uniform to uniform case is:

$$\vartheta(x_0,x) = \frac{N(x_0,x)}{N_u(x_0,x)} = \frac{g(x_o)}{x - x_o} \int_{x_o}^{x} \frac{d_u}{g(u)}$$

Similarly, the inverse is:

$$\vartheta'(x_0,x) = \frac{N_u(x_0,x)}{N(x_0,x)} = \frac{x - x_o}{g(x_i) \int_{x_o}^{x} \frac{d_u}{g(u)}}$$

For the whole partition these become:

$$\vartheta(X_j, X_{j+1}) = \frac{g(X_j)}{X_{j+1} - X_j} \int_{X_j}^{X_{j+1}} \frac{dx}{g(x)}$$

and

$$\vartheta(x_j, x_{j+1}) = \left[\frac{g(x_j)}{x_{j+1} - x_j} \int_{x_j}^{x_{j+1}} \frac{dx}{g(x)} \right]^{-1}$$

where

$$g(x) = \frac{1}{\left[[f^{(m+1)}(x)]^2 w(x) \right]^{\frac{a\ 1}{2m+3}}}$$

For the interval [a,b], ϑ becomes:

$$\vartheta(a,b) = \frac{g(a)}{b-a} \int_a^b \frac{dx}{g(x)}$$

Note that this ratio, although a first order approximation, does not depend on:

- the value of ε_o
- the type of polynomial (Chebyshev, Legendre . . .)

but depends on:

- the behavior of f(x) and its (m+1)st derivative $f^{(m+1)}(x)$.
- the degree m of the polynomial approximant.

Note also that N(a,b) = - N(b,a) and that N(a,b) = N(a,c) + N(c,b).

Example:

Using $f(x) = 1/x$ as an example with $m = 1$,

$$\vartheta_{(x_j, X_{j+1})} = \frac{1}{X_{j+1} - X_j} \int_{X_j}^{X_{j+1}} \left(\frac{X_j^3}{x^3} \right)^{\frac{2}{5}} dx$$

With $X_j = .5$ and $X_{j+1} = 1$, one obtains

$$(.5, 1) = .603$$

The results obtained by computer program are in good agreement with this value for intervals less than 2^{-6}. See Table 3.1.

In summary, we have found a way to compute the number of breakpoints in the interval $[X_j, X_{j+1}]$. This expression is necessary for computing the NLP constraint on the number of ROM words.* In the next section we briefly consider the case where the non-linear programming problem can be simplified or linearized.

---

*The limit on $\vartheta(X_j, X_{j+1})$ when $m \to \infty$ is treated in appendix A.3.

h	# of ROM Words Non-Uniform Interval Length	# of ROM Words Uniform Interval Length	(.5,1) Ratio
2^{-4}	6	8	.75
2^{-5}	11	16	.69
2^{-6}	20	32	.625
2^{-7}	40	64	.625
2^{-8}	78	128	.61
2^{-9}	156	256	.61
2^{-10}	311	512	.609
2^{-11}	621	1024	.606
2^{-12}	1241	2048	.606

For $f(x) = 1/x$ between .5 and 1.

P_m is a Chebyshev polynomial of degree 1.

Table 3.1

Number of ROM Words for Uniform and Non-Uniform Intervals

3.3.3 Linearization of the Non-Linear Programming Problem

As defined in Section 3.3.1 our problem is to solve for the (s-1) joints, $X_2, \ldots, X_s$:

$$\text{Minimize} \ldots \left\{ \prod_{j=1}^{s} \left(\frac{X_{j+1}}{X_j} \right)^{m_j} \right\} \qquad \text{with } X_1 = a = \frac{1}{\beta} \text{ and } X_{s+1} = b = 1.$$

subject to the constraint:

$$\sum_{j=1}^{s} (m_j+1) N\ (X_j, X_{j+1}) \leqslant R_0$$

where we compute $N\ (X_j, X_{j+1})$ as explained in the last section.

It is possible to linearize this NLP as follows:

If the distribution of mantissas of floating point numbers is assumed to be uniform in $[1/\beta, 1]$ then, minimizing the average number of multiplications

$$M_{av} = \int_{X_1}^{X_{S+1}} \frac{\beta - 1}{\beta} M(x)\,dx$$

becomes:

$$\min \left\{ \sum_{j=1}^{s} (m_j+1)(X_{j+1}-X_j) \right\}$$

which is a linear objective function.

To linearize the constraint, one can assume a UIVD realization and within each interval $[X_j, X_{j+1}]$ the number of breakpoints is

$$N(X_j, X_{j+1}) = \frac{X_{j+1}-X_j}{h_j}$$

where h_j is the minimum breakpoint width guaranteeing a precision ε_o over the whole interval $[1/\beta, 1]$ when using approximating polynomials of degree m_j. The constraint:

$$\sum_{j=1}^{s} (m_j+1) \left(\frac{X_{j+1}-X_j}{h_j} \right) \leqslant R_0$$

is linear. The other constraints:

$$X_j - X_{j+1} \leq 0 \qquad j = 1, \ldots, s$$

are also linear.

The problem has now been transformed into a linear programming problem of the form:

$$\text{minimize } \underline{C}^T \underline{X}$$

$$\text{subject to } \underline{\underline{A}}\ \underline{X} \leq \underline{b}$$

where $\underline{X}$ is the (s-1) dimensional column vector:

$$\underline{X} = (x_2, \ldots, x_j, \ldots, x_s)^T$$

and $\underline{C}$ is the (s-1) dimensional column vector:

$$\underline{C} = (C_2, \ldots, C_d, \ldots, C_s)^T$$

$$\text{with } c_d = (m_{j-1} - m_j) \quad j = 2, \ldots, s$$

$\underline{\underline{A}}$ is the (s) x (s-1) matrix:*

$$\underline{\underline{A}} = \begin{array}{|cccc|} a_2 & a_3 & \ldots & a_s \\ -1 & & & \\ 1 & -1 & & \\ & +1 & -1 & \\ & & \ddots & \\ & & & +1 \quad -1 \\ & & & 1 \end{array} \quad \begin{matrix} \\ \\ \\ \\ \\ \\ \\ \end{matrix} \quad \begin{matrix} \\ \\ \\ \\ \\ \\ \\ \end{matrix}$$

(The O's in the triangular regions above and below the diagonal are zeros.)

s

(s-1)

*Note that A has an interesting diagonal form and that Ax=b can be solved without actually inverting A.

where the elements

$$a_j = \left(\frac{m_{j-1}+1}{h_{j-1}} - \frac{m_j+1}{h_j} \right)^T$$

and $\underline{b}$ is the s-dimensional column vector:

$$\underline{b} = \left(R_0 - \frac{X_{s+1}}{h_s} + \frac{X_1}{h_1}, -X_1, 0, \ldots, X_{s+1} \right)^T$$

The form of this problem can be put into standard linear program form by the use of slack variables and the answers would be the positions of the joints X_j, j=2, ..., s.

This linearization may be of use for cases involving a large number of joints but we have limited ourselves to 2 joints and the NLP problem will be solved without linearization.

Quadratic case

For the quadratic case, the function to be minimized can be approximated by a quadratic function of the joint locations:

$$\prod_{j=1}^{s} \left(\frac{X_{j+1}}{X_j} \right)^m = \sum_{i=1}^{s} a_i X_i^2 + \sum_{i \neq j}^{s} b_{ij} X_i X_j + \sum_{j=1}^{s} c_j X_j + d$$

The problem is then transformed into a quadratic problem. A method like the steepest descent could be successfully applied to search for a minimum.

4. THE SOLUTION OF THE NON-LINEAR PROGRAMMING PROBLEM FOR $f(x) = 1/x$

The non-linear programming problem to be solved (as described in Section 3) falls into the general class known as constrained, medium-scale problems. For our purposes, the number of internal joints has been limited to one and two. A computer solution is essential in cases like this if easily interpretable results are to be obtained. In addition, this choice of solution was motivated by the existence of reliable software for solving the general minimization problem. A large number of different algorithms are in fairly common use but the programs implementing these algorithms are not readily available and have generally been used on a limited basis only. Since our goal was not to write a program for solving the general problem of constrained minimization, some other method had to be found which alleviated writing an extensive and very involved software package. Fortunately, there exists reliable and readily accessible software for computing the unconstrained minimum of a function of N variables. Three FORTRAN programs that are readily available are:

FMFP - (IBM Scientific Subroutine Package)

This subroutine finds a local minimum of a function of several variables by the method of Fletcher and Powell [IBM68].

FMCG - (IBM Scientific Subroutine Package)

This subroutine finds a local minimum of a function of several variables by the method of conjugate gradients [IBM68].

ZXMIN - (International Mathematical Software Library)

This subroutine determines the minimum of a function of N variables using a quasi-Newton method.

In order to simplify the problem, we were led to consider methods that replace, or approximate, constrained optimization problems by unconstrained problems. Such methods exist, are readily available, and are known as Penalty and Barrier function methods. These methods are discussed below and used, ultimately, to solve the optimization problem.

4.1 Penalty Function Method

The basic idea of Penalty/Barrier methods is to add to the objective function a term that prescribes a high cost

for violation of the constraints (Penalty Method) or to add a term that favors joints interior to the feasible region over those near the boundary (Barrier Method). This high cost is represented by a parameter μ that weights the size of the Penalty or Barrier. When $\mu \to \infty$ the approximation to the original problem becomes more and more accurate and tends toward the true solution.

These methods are appealing because:

- They are comparatively simple and straightforward to implement.
- The software exists and consists of a routine to find the unconstrained minimum of a function of N variables.
- They possess a certain degree of generality.

However, as the parameter μ is increased so as to yield a good approximation to the constrained problem, the corresponding structure of the unconstrained problem becomes increasingly unfavorable in that the convergence rate of the algorithms used to find the minimum decreases. A study of the location of the approximate solution (optimum) as a function of μ has been made without knowing the exact behavior of these algorithms as a function of μ.

The programs used were implemented in FORTRAN. The study was limited to the one and two joint cases, that is, the one and two variable NLP problem. Only the Penalty Method was used.

The problem, then, may be stated as follows:

$$\left.\begin{array}{l}\text{minimize } M(\underline{X}) \text{ (objective function)} \\ \text{subject to } \underline{X} \in S \text{ (constraints)}\end{array}\right\} \quad (4\text{-}1)$$

where M is a continuous function on E^n and S is a constraint set in E^n. In our case, S is a set of inequalities describing the relative positions of the joints $\underline{X} = (X_j, X_{j+1} \ldots)$ and the inequality concerning the number of available Read Only Memory words R_0. The penalty function method replaces problem (4-1) by an unconstrained problem of the form:*

$$\text{minimize } Q(X, \mu) = M(\underline{X}) + \mu P(\underline{X})$$

where μ is a positive parameter and the penalty function P is a function of the vector $\underline{X}$ and is defined as follows:

$$P(\underline{X}) \qquad \begin{array}{l}\text{(i) P is continuous} \\ \text{(ii) } P(X) \geq 0 \text{ for all } \underline{X} \in E^n \\ \text{(iii) } P(\underline{X}) = 0 \text{ if an only if } \underline{X} \in S\end{array}$$

In other words, the new problem minimizes the objective function only when $\underline{X}$ satisfies the constraints $\underline{X} \in S$ and minimizes the objective function plus a penalty whenever $\underline{X}$ violates the set of constraints S.

---

*The meanings of P, Q and S here are different from those used in Sections 2 and 3.

The problem, then, is to devise a penalty function $P(\underline{X})$ that is continuous and positive but null whenever $\underline{X}$ is within S. The penalty function is weighted by the parameter μ which is allowed to increase toward ∞ so as to approach the exact solution of the initial problem (4-1).

The procedure for solving problem (4-1) is as follows:*

Let $\{\mu_k\}$, $k = 1,2,\ldots$ be a sequence tending to infinity such that for each k, $\mu_k \geq 0$ and $\mu_{k+1} > \mu_k$ Then, for each k, solve the problem:

$$\text{minimize } Q(\mu_k, \underline{X}_k) = M(\underline{X}_k) + \mu_k P(\underline{X}_k) \qquad (4\text{-}2)$$

obtaining a solution $\underline{X}_k$. It is assumed that a solution to (4-2) exists for every k.

The following two lemmas [LUE72] lead to the global convergence of the approximate solution $\underline{X}_k$ to the exact solutuion of the initial problem.

Lemma 1

$$\begin{cases} Q(\mu_k, \underline{X}_k) \leqslant Q(\mu_{k+1}, \underline{X}_{k+1}) \\ P(\underline{X}_k) \geqslant P(\underline{X}_{k+1}) \\ M(\underline{X}_k) \leqslant M(\underline{X}_{k+1}) \end{cases}$$

where $\underline{X}_k$ is a solution of (4-2).

---

*The use of k and μ here is different from that of Section 2.

Lemma 2

Let $\underline{X}^*$ be a solution of problem (4-1). Then for each k,

$$M(\underline{X}^*) \geqslant Q(\mu_k, \underline{X}_k) \geqslant F(\underline{X}_k)$$

Theorem. Let $\underline{X}_k$ be a sequence generated by the penalty method. Then any limit Point P of the sequence is a solution to (4-1).

Next we will devise a penalty function that verifies the three properties stated above.

As shown in Section 3.3.1, the problem can be written:**

$$\text{minimize} \quad M_{av}(\underline{X}) = \ln \left\{ \left(\frac{X_2}{a}\right)^{m_1} \left(\frac{X_3}{X_2}\right)^{m_2} \left(\frac{X_4}{X_3}\right)^{m_3} \cdots \left(\frac{b}{X_s}\right)^{m_s} \right\}$$

where $X_1 = a$ and $X_{s+1} = b$ are the interval boundaries.

We assume here that $1/\beta \leq a \leq b \leq 1$. The constraints are:

$$\begin{cases} \sum_{j=1}^{s} N_j(X_j, X_{j+1})(m_j+1) \leqslant R_0 \\ X_1 \leqslant X_2 \\ X_2 \leqslant X_3 \\ \vdots \\ X_s \leqslant X_{s+1} \end{cases} \tag{4-4}$$

**Between $1/\beta$ and 1 the probability density of floating point mantissas is $1/x\log\beta$. The case where this does not apply is considered in the Appendix.

In this case, S is defined by a certain number of inequality constraints of the form:

$$S = \{ \underline{X} : g_j(\underline{X}) \leq 0 \quad j = 1,\ldots,s+1\}$$

The functions $g_j(\underline{X})$ are:

$$\begin{cases} g_{s+1}(\underline{X}) = \sum_{j=1}^{s} (m_j+1)N_j(X_j,X_{j+1})-R_0 \\ g_1(\underline{X}) = X_1-X_2 \\ g_2(\underline{X}) = X_2-X_3 \\ \vdots \\ g_s(\underline{X}) = X_s-X_{s+1} \end{cases} \tag{4-5}$$

A penalty function that satisfies properties (i), (ii), and (iii) and which is frequently used is:

$$P(\underline{X}) = \sum_{j=1}^{s+1} (\max(0,g_j(\underline{X})))^2$$

since if $g_j(X) \leq 0$ for $j = 1,2,....s$, then $P(\underline{X}) = 0$ but $P(\underline{X})$ is always positive if $g_j(\underline{X}) > 0$ for some j. Consequently, we must now implement the function

$$Q(\mu, \underline{X}) = M_{av}(\underline{X}) + \mu P(\underline{X})$$

in FORTRAN and then link it with an existing minimization routine in order to solve our general NLP problem.

4.2 Software Considerations

The routine chosen to compute the minimum was ZXMIN of IMSL. The IMSL software is well supported unlike the IBM SSP routines. The storage requirements, although not critical for our application, are smaller for ZXMIN (the Hessian Matrix is stored in symmetric storage mode). Moreover, the routine ZXMIN implements Newton's method which has the advantage of alleviating some of the slow convergence problems associated with the extremely unfavorable eigenvalue structure that accompanies penalty methods. The subroutine ZXMIN also evaluates the Hessian Matrix and the user is not required to supply formulas for its evaluation.

A FORTRAN program has been written to

a. evaluate the number of ROM words for a given joint configuration.

b. determine the average number of multiplications using the $1/x \ln \beta$ probability density function.

The above penalty function was used and the routine ZXMIN was modified for this specific problem. The parameters that can be fixed by the user are

- ε_o, the precision of the evaluation of f(x)
- equal vs unequal intervals
- number of joints, and the
- degree of polynomial in each partition.

The polynomials were assumed to be of the Chebyshev type and the colloquation points were computed accordingly.

The weight function w(x) was implemented as a function subroutine and, for simplicity, $w(x) \equiv 1$ was chosen.

The structure of the program is shown in Fig. 4.1. The program was run under the FORTRAN IV G compiler. For a listing of IMSL routines, refer to IMSL Corporation. Routine ZXMIN is not given in the appendix because it is copyrighted. The remainder of the program is listed in Appendix A.1.

Here we will describe each subroutine from the bottom up along with a brief explanation of the meaning of the parameters. First recall that in the general case, we must solve for the length of each breakpoint interval $[x_i, x_{i+1}]$ within the partition $[X_j, X_{j+1}]$ by solving the following equation for h_i.

$$\int_{x_i}^{x_i+h_i} (f(x)-P_m(x,h_i))^2 w(x)\,dx = \epsilon_0^2. \tag{4-6}$$

We have seen (Section 3.3.2) that the above equation can be replaced by:

$$h^{2m+3}\left[\frac{f^{(m+1)}(x_i)}{(m+1)!}\right]^2 \int_0^1 \prod_{k=0}^{m}(r-\alpha_k)\,w(x_i+rh)\,dr = \epsilon_0^2 \tag{4-7}$$

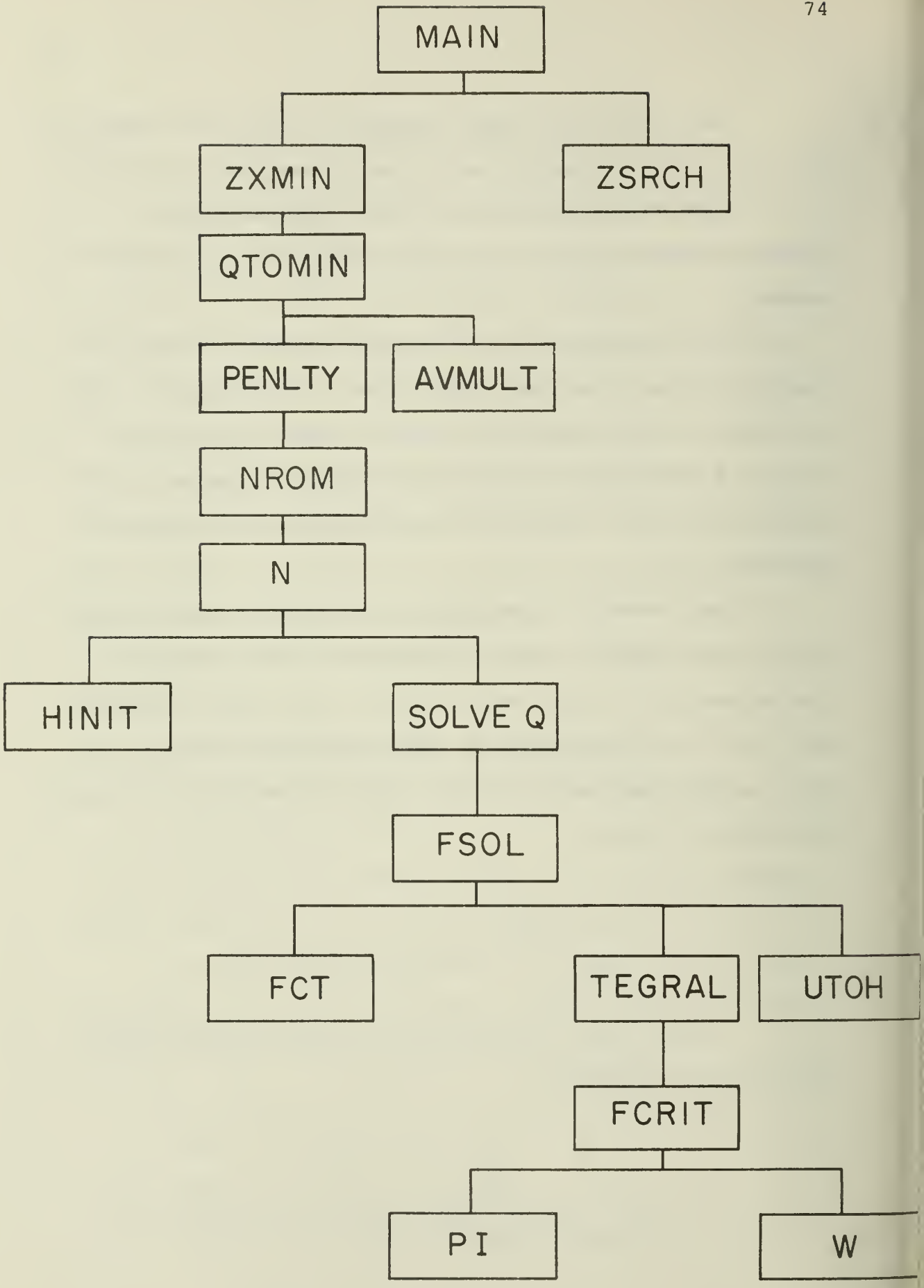


Figure 4.1 Computer Solution; Subroutine Structure

This avoids many rounding problems associated with the computation of the difference

$$f(x) - P_m(x, h)$$

which can become very small.

The following routines are required. (See Fig. 4.1 and Appendix A.1.)

Function W(XV,NTYPE)

Parameter XV is the x variable of equation (4-6). Parameter NTYPE is the order of the derivative of w(x). If one wishes to use the relative error in equation (4-6), simply replace w(x) by $\frac{w(x)}{(f(x))^2}$

Function PI(XV,M)

PI represents the Lagrangian polynomial:

$$\frac{\prod_{k=0}^{m_j} (x-x_k)}{(m_j+1)!} \qquad (4\text{-}8)$$

where the x_k's are the roots of the Chebyshev Polynomial of degree m_j in the interval $[x_i, x_{i+1}]$. The roots of the normalized polynomial of degree m_j are referred to as α_k. Parameter M is the degree of polynomial (4-8), m_j.

Function FCT(XV,M)

FCT represents function f(x) in (4-6). Parameter M represents the order of the derivative of function f. If M = 0, the function returns the value of f(XV). Otherwise the derivative is computed by the subroutine. The derivative function was hard-coded due to the simplicity of the functions used ($\frac{1}{x}$, lnx, . . .)

Function FCRIT (R, HU, INTKND, XI, M, PI, W)

FCRIT is the product:

$$\left[\frac{\prod_{k=0}^{m_j}(r-\alpha_k)}{(m_j+1)!}\right]^2 w(x_i+rh) \quad \text{when INTKND=1}$$

(See 4-7)

or the product

$$\left[\frac{\prod_{k=0}^{m_j}(r-\alpha_k)}{(m_j+1)!}\right]^2 r\cdot w'(x_i+rh) \quad \text{When INTKND = 2}$$

or

$$\left[\frac{\prod_{k=0}^{m_j}(r-\alpha_k)}{(m_j+1)!}\right]^2 \quad \text{for INTKND = 3}$$

M is, of course, m_j while the jth partition is being considered.

These 3 functions appear in the solution of (4-7) using Newton's method.

Parameter R: represents variable r and takes values between 0 and 1.

Parameter HU: represents h, the width of $[x_i, x_{i+1}]$

Parameter XI: represents x_i

Function TEGRAL (XL, XU, HU, INTKND, XI, M, PI, W)

Function TEGRAL computes the integral of FCRIT summed over R from XL to XU. The evaluation is done by means of a 24-Point Gauss Quadrature formula which integrates polynomials up to degree 47 exactly.

Subroutine FSOL (U, F, DERF, XI, M, PI, FCT, W, ETA, EPSO)

The use of function FSOL can be explained as follows:

Since h and ε_o are very small, it seemed more convenient to rearrange equation (4-7) as:

$$\frac{(f^{(m+1)}(x_i))^2}{((m+1)!)^2}\int_0^1 \prod_{k=0}^{m} (r-\alpha_k)^2\, w(x_i+rh)\, dr = \frac{\epsilon_o^2}{h^{2m+3}}$$

Here m ≡ mj. Then, the change of variables:

$$u = \frac{h^{m+2}}{\varepsilon_o}$$

was made and the functions FBIG and FSMALL were defined as:

$$FBIG = \frac{(f^{(m+1)}(x_i))^2}{((m+1)!)^2} \int_0^1 \prod_{k=0}^{m} (r-\alpha_k)^2 \, w(x_i + rh)dr$$

$$FSMALL = \frac{h}{u^2}$$

Then the equation:

$$FBIG(u) - FMSMALL\ (u) = 0$$

was solved for u using Newton's method:

$$u_n = u_{n-1} - \frac{FBIG(u_n) - FSMALL(u_{n-1})}{(FBIG(u_{n-1}) - FSMALL(u_{n-1}))'}$$

Therefore the derivative of FBIG - FSMALL is needed:

$$(FBIG)' = \frac{(f^{(m+1)}(x_i))^2}{((m+1)!)^2} \int_0^1 \prod_{k=0}^{m} (r-\alpha_k)^2 r\, w'(x_i+rh)dr \, \frac{dh}{du}$$

$$(FSMALL)' = -\frac{(2m+3)}{(m+2)} \frac{h}{u^3}$$

with $\frac{dh}{du} = \frac{h}{(m+2)u}$

Function FSOL returns the values of FBIG - FSMALL and its derivative. The subroutine UTOH returns U as a function of H.

Function SOLVEQ

This function solves FBIG = FSMALL for U as defined above using Newton's method. To find the solution for H the routine UTOH is called. The starting point of the iteration is provided by the subroutine HINIT.

Function N(IUNIF, XI, X2, M, PI, FCT, W, ETA, EPSO, IER)

This function computes the number of breakpoints between Xl and X2. It calls SOLVEQ repeatedly and performs the iteration

$$x_{i+1} = x_i + k_m g(x_i)$$

until x_i exceeds X_2.

The parameter IUNIF provides the option of computing N using uniform or variable intervals.

Function NROM (IUNIF, NJOINT, XJ, JDEGR, PI, FCT, W, ETA, EPSO)

This function returns the value of the number of Read-Only Memory words for implementations with NJOINTs and degrees of polynomials JDEGR(I), I = 1,2 . . . NJOINT.

The values of the joints are in the array XJ and are an input to the routine.

Function PENLTY (NORK, IUHIF, NJOINT, XJ, JDEGR, PI, FCT, W, ETA, EPSØ)

This function computes the penalty function as defined above. The maximum number of ROM words is passed in the parameter NROK. The function calls NROM with the joint parameters to determine the number of ROM words.

Function AVMULT (NJOINT, XJ, JDEGR, BASE)

This determines the average number of multiplications assuming a logarithmic distribution:

$$\frac{1}{\text{XLOG(BASE)}}$$

Function QTOMIN (XMV, NROK, IUNIF, NJOINT, XJ, JDEGR, BASE, PI, PCT, W, ETA, EPSO)

This is the function Q to be minimized and whose minimization determines the optimal location of the joints. The function implemented is:

QTOMIN = AVMULT + XMU * PENLTY

ZXMIN and ZSRCH are IMSL routines to minimize a function of n variables and to perform a multi-dimensional

grid search. ZSRCH was not used, however, due to the nature of the 2 joint case studied here.

4.3 Study of the Minimum

In what follows we limit our discussion to the one and two joint cases. These two cases illustrate all aspects of the search for a solution to the NLP problem and introducing more joints is just an extension of the results presented here. In addition, two joints is sufficient for all of the single precision, interesting solutions for $f(x) = 1/x$.

A qualitative description of the surface and the location of the minimum is necessary in order to understand the plots that follow. Recall that we are attempting to minimize:

$$Q(\mu, \underline{X}) = M_{av}(\underline{X}) + \mu \cdot P(\underline{X})$$

Assume that there are two joints, or three different polynomials having degrees 1, 2 and 3. The function whose minimum is to be found is represented, in this case, by a surface when plotted as a function of the joint's abscissas between .5 and 1.

When X_2 (denoted XJ(2) on the plots) moves from .5 to 1, the interval width [.5, XJ(2)] increases and therefore the number of multiplications decreases whereas the number

of breakpoints $[x_i, x_{i+1}]$ increases. For XJ(2) fixed, moving XJ(3) has the same effect, but it is less pronounced because of the higher degree polynomial. These facts are illustrated in Fig. 4.2. In Fig. 4.2 the average number of multiplications (the objective function) is plotted as a function of XJ(2) and XJ(3). Because XJ(2) must be $\leq$ XJ(3), only half of the X_2X_3 plane has data. This is in accordance with the fact that the second joint is always to the right of the first one. In Fig. 4.3, 4.4, and 4.5, the number of ROM words is plotted as a function of XJ(2) and XJ(3). Note that X_2 and X_3 axes are interchanged relative to Fig. 4.2. Notice here that for XJ(3) fixed, the variations of the number of ROM words as a function of XJ(2) is relatively steep. The reason for this is that by moving XJ(2), one adds intervals corresponding to degree 1 approximations. They grow rapidly in number because these intervals are small. Next, Fig. 4.4 and 4.5 show that XJ(3) creates similar, but less pronounced, variations in the number of ROM words since the degree is higher and therefore the number of intervals required changes more slowly.

The combination objective + penalty creates the surface sketched in Fig. 4.6. One can make the following observations:

1. Points D_1, D_2, D_3... are situated along the diagonal of the $D_1O_1O_2$ plane on the X_2X_3 diagonal. To

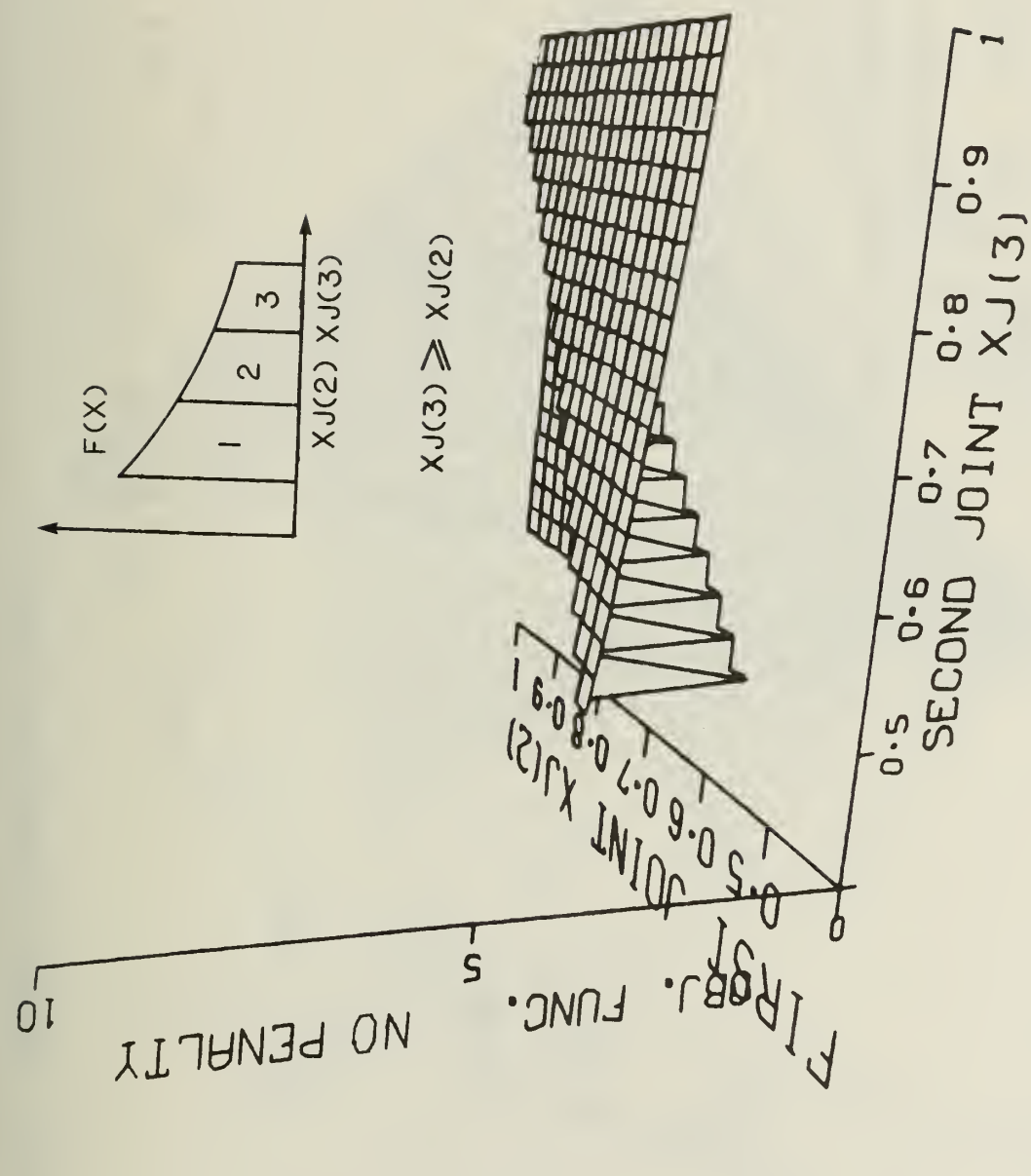


Figure 4.2 Average Number of Multiplications; No Penalty

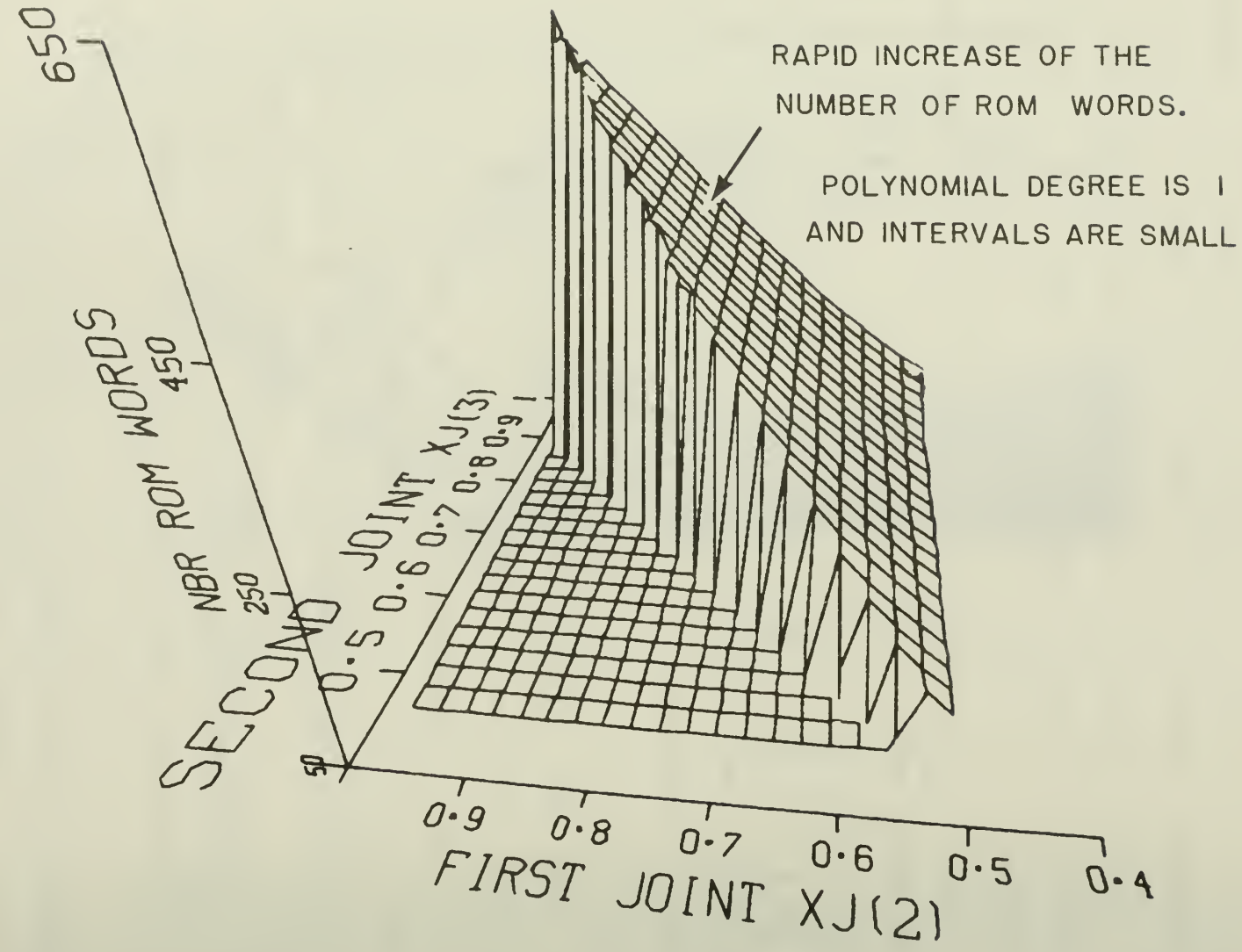


Figure 4.3 Number of ROM Words (View 1)

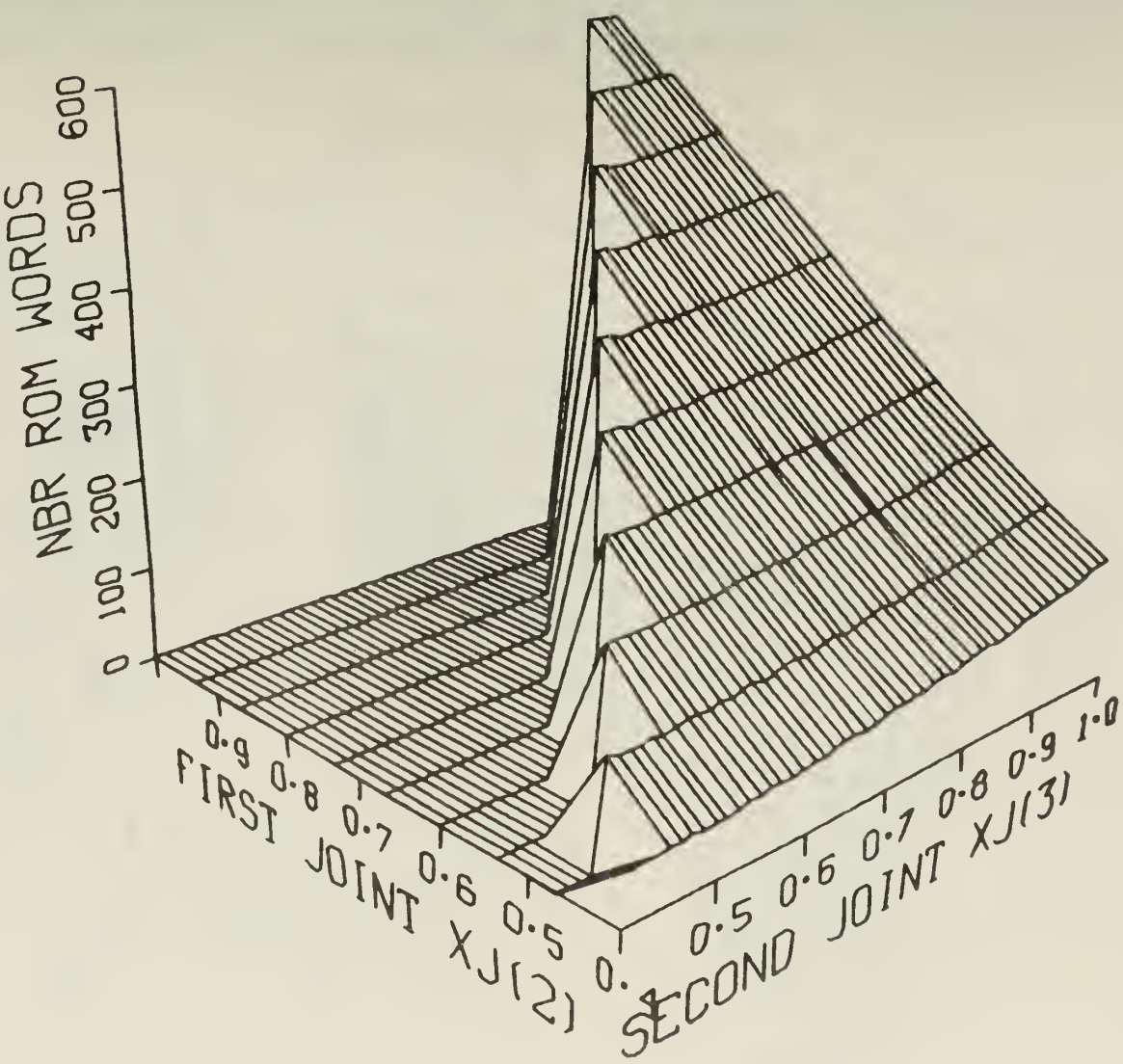


Figure 4.4 Number of ROM Words (View 2, Rotation)

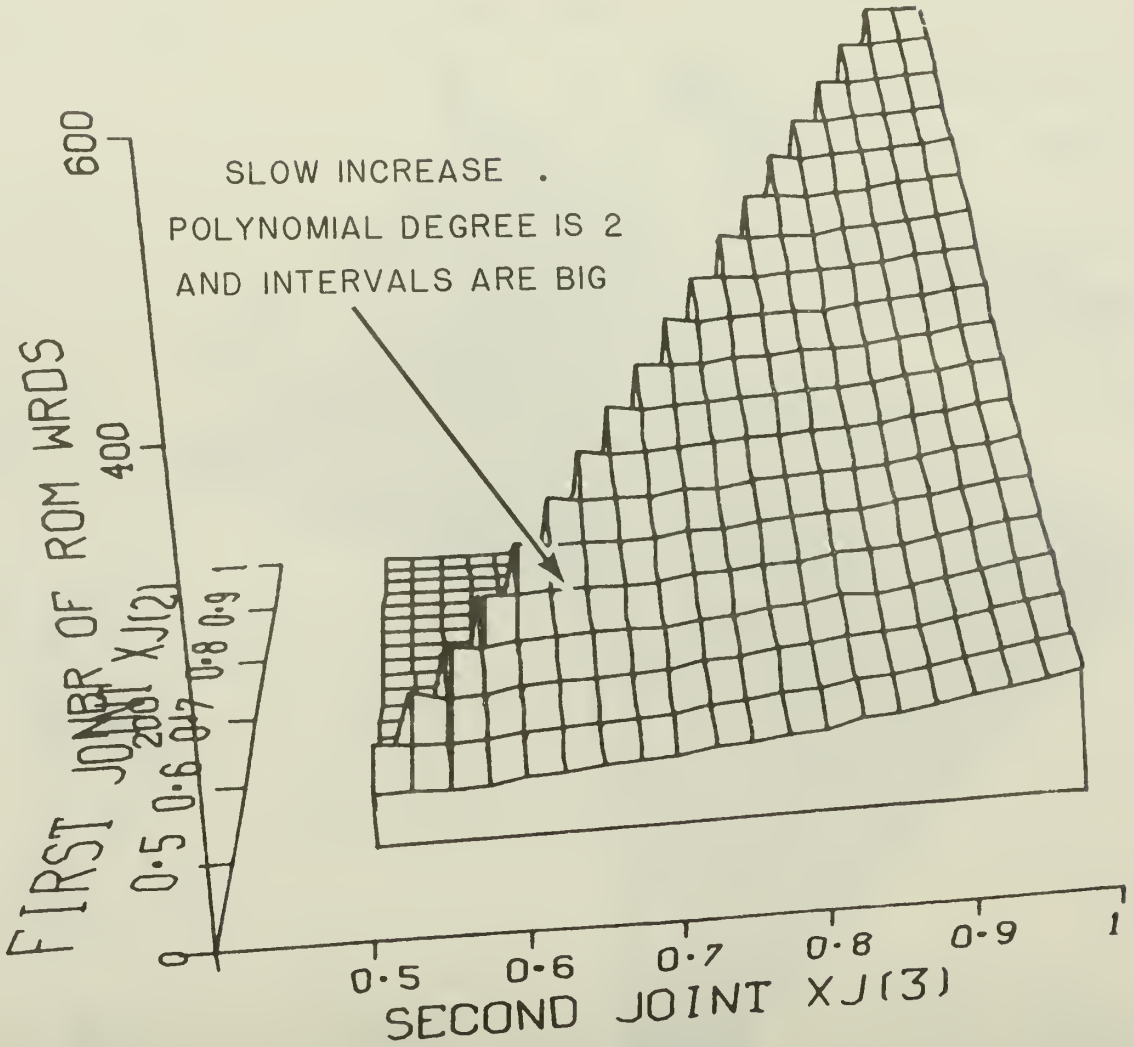


Figure 4.5 Number of ROM Words (View 3, Rotation)

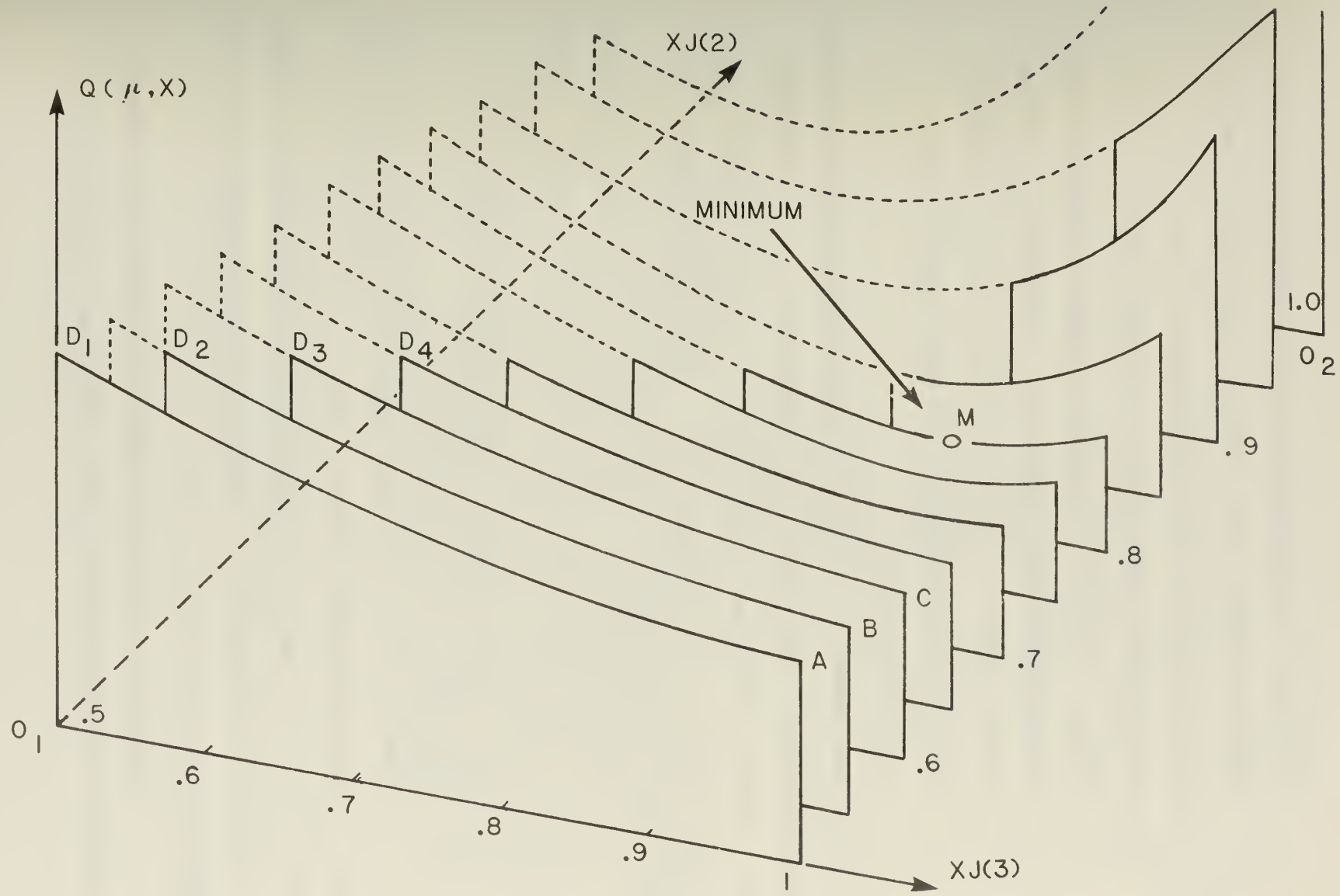


Figure 4.6 Location of the Minimum on the Surface Representing Objective + Penalty as a Function of Joints Locations

respect the constraints on the relative position of the joints, the curves of interest are on the $X_3 \geq X_2$ side of the diagonal.

2. A is higher than B which is higher than C: when XJ(2) increases the average number of multiplications decreases. Of course, the number of words of ROM required increases but does not influence the curve as long as it is less than R_0, the allowed maximum.

The penalty starts applying when the number of ROM words exceeds a given value R_0 (around XJ(2) = .65 on Fig. 4.6). Before the penalty applies, the curves represent the average number of multiplications. The region marked M is where the minimum occurs. The minimum gives

a. a value for XJ(2),

b. a value for XJ(3),

c. the average number of multiplications, and

d. the number of ROM necessary, R_0.

From D_1 to A the penalty does not apply yet and the curve D_1A represents the average number of multiplications which decreases when X_3 increases because this has the effect of replacing polynomials of degree m_3 by polynomials of lower degree m_2.

The next curve D_2B is below the curve D_1A. More precisely, for equal X_3's D_2B is less than D_1A. This is

because X_2 has increased and consequently more degree (m_2) polynomials have been traded for polynomials of lower degree m_1 and the average number of multiplications has diminished.

Figure 4.7 can be considered as a combination of Fig. 4.2 (average number of multiplications) and Fig. 4.5 (number of ROM words). By taking the "combination"

(Fig. 4.2) + μ · Penalty (Fig. 4.2)

one has a visual illustration of the origin of the surface representing Q.

4.3.1 Influence of μ

Figures 4.8 through 4.12 represent the influence of μ on the position of the minimum. μ increases from 0 to 10^6 and a value of μ of the order 10000 was chosen for the runs without creating stability problems in solving the problem on the computer. Illustrated is the fact that μ weights the penalty function and this contributes to the steepness of the "penalty wall."

4.3.2 Influence of the Maximum Number of ROM Words R_0

By increasing the number of available Read Only Memory words, one can intuitively expect the following:

If R_0 is increased, more polynomials can be used.

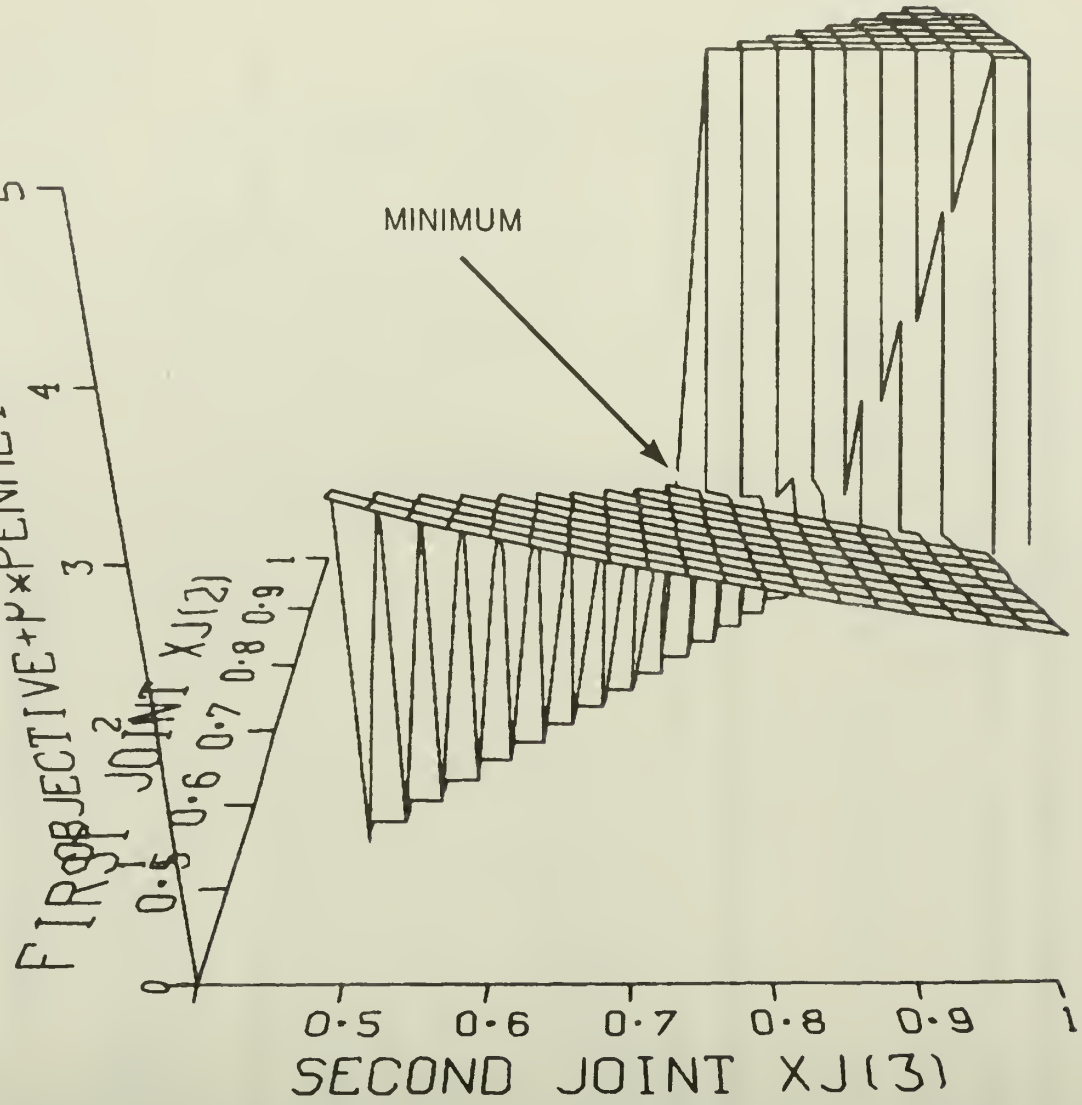


Figure 4.7 Combination of Average Number of Multiplications and the Penalty on the Number of ROM Words

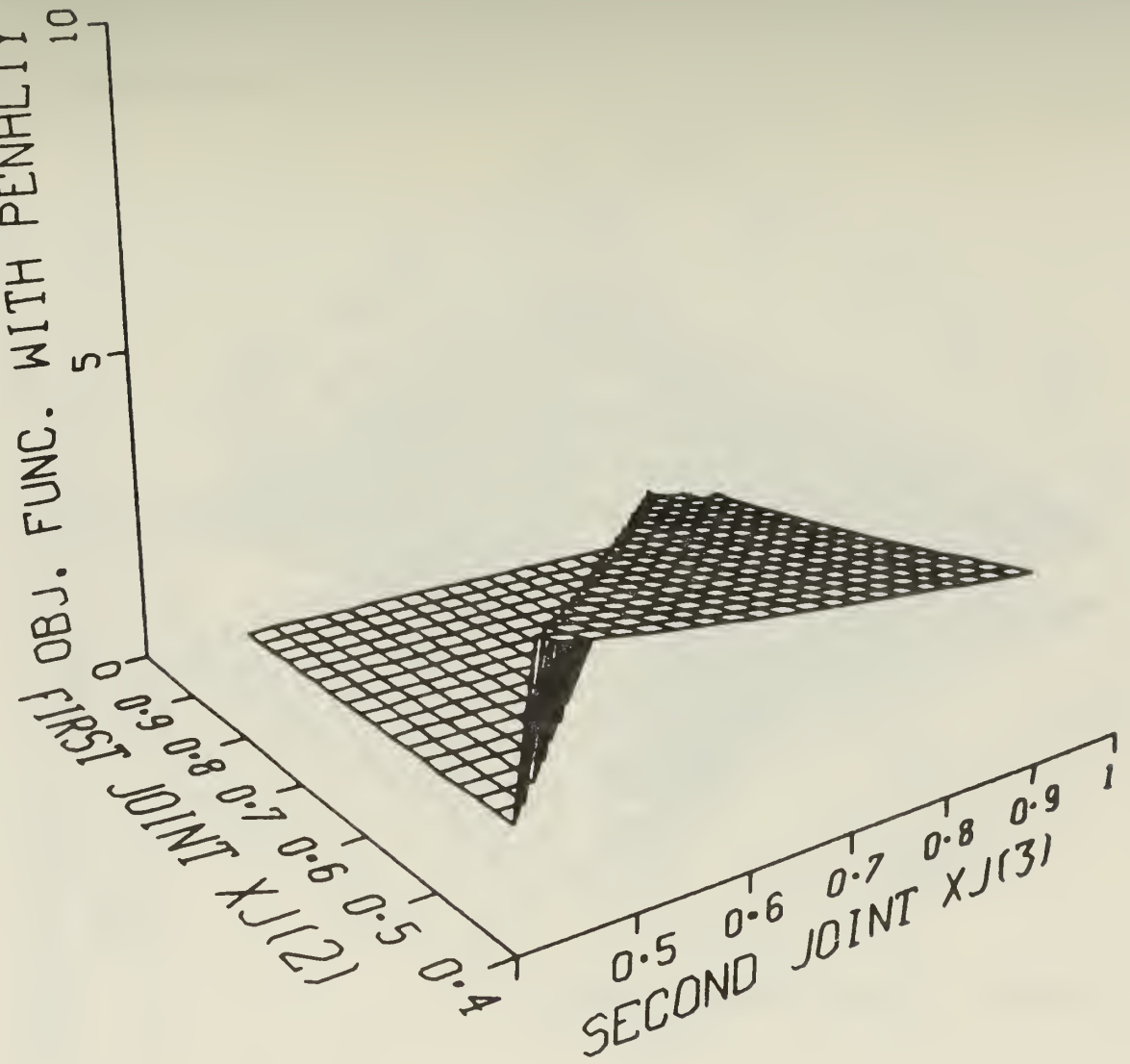


Figure 4.8 Influence of μ; $\mu = 0$

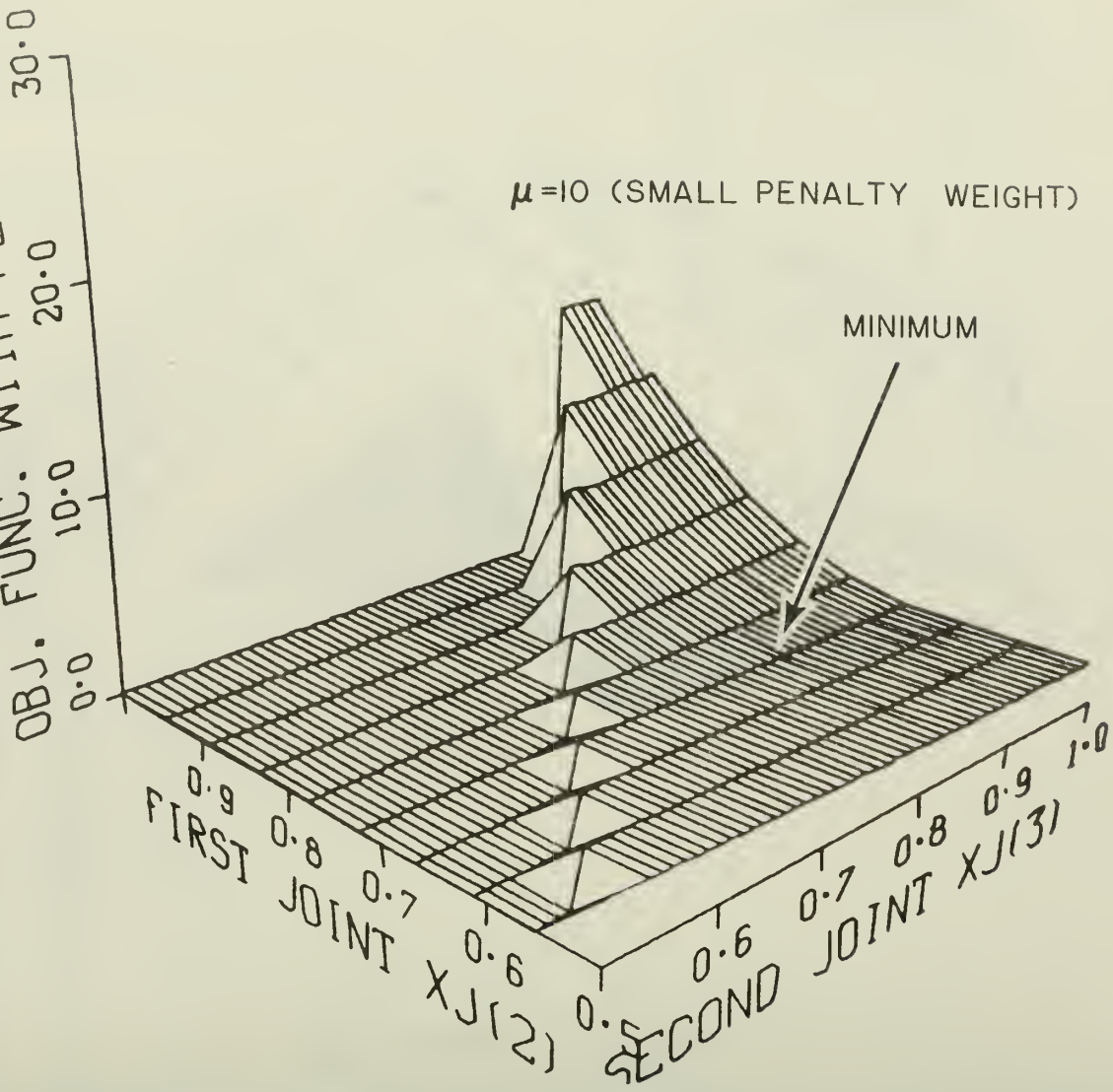


Figure 4.9 Influence of μ; μ = 10

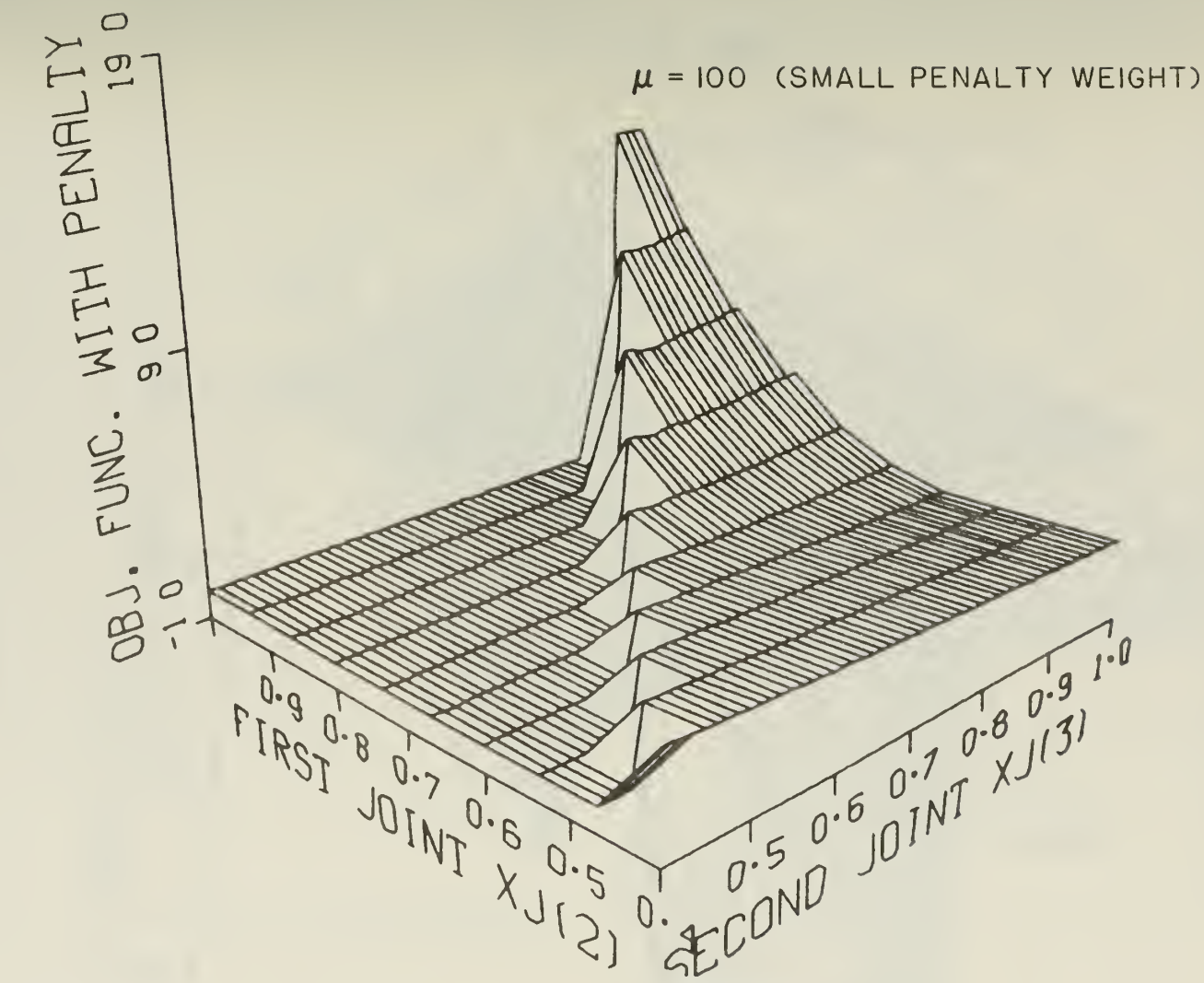


Figure 4.10 Influence of μ; μ = 100

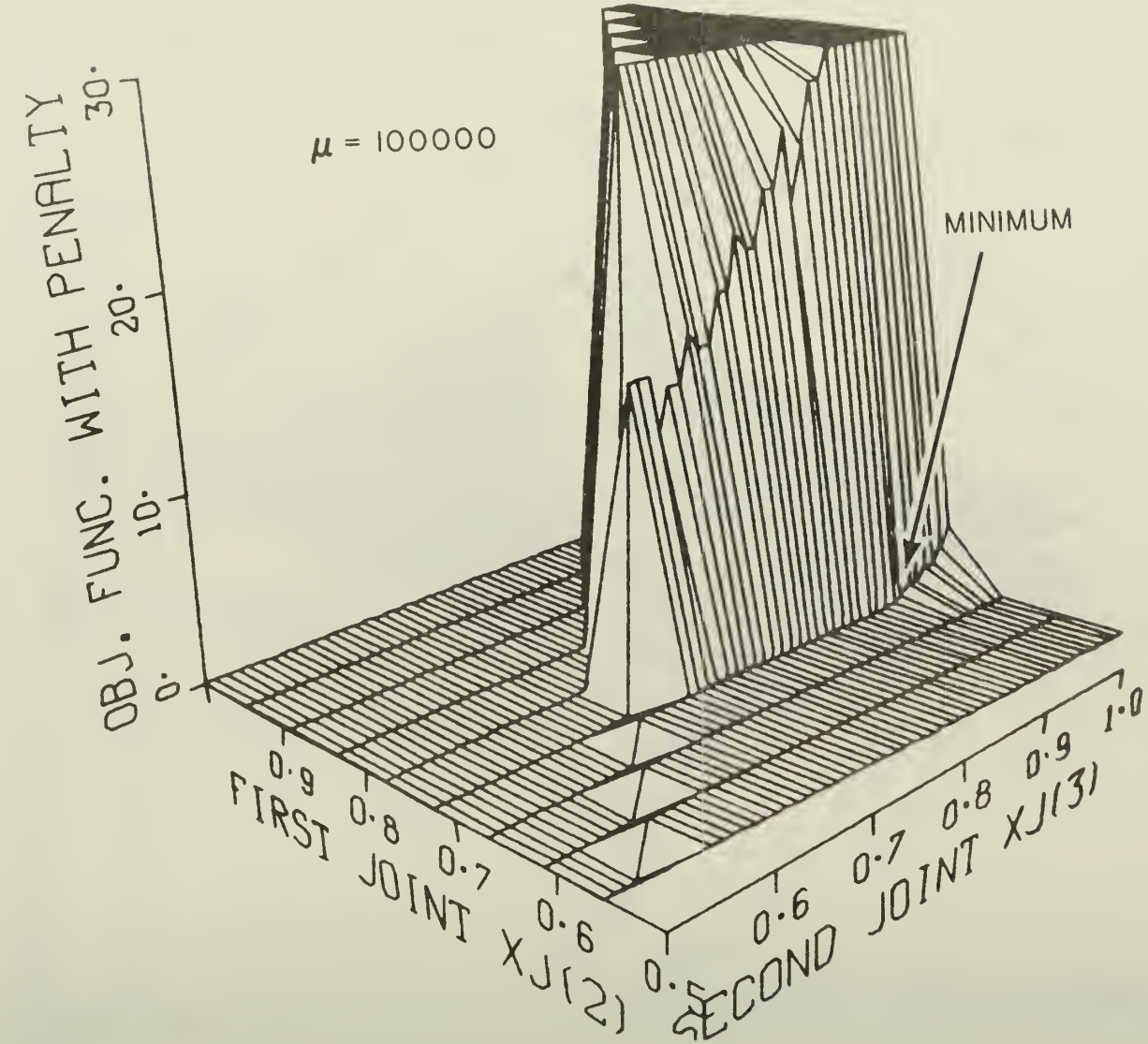


Figure 4.11 Influence of μ; μ = 10,000

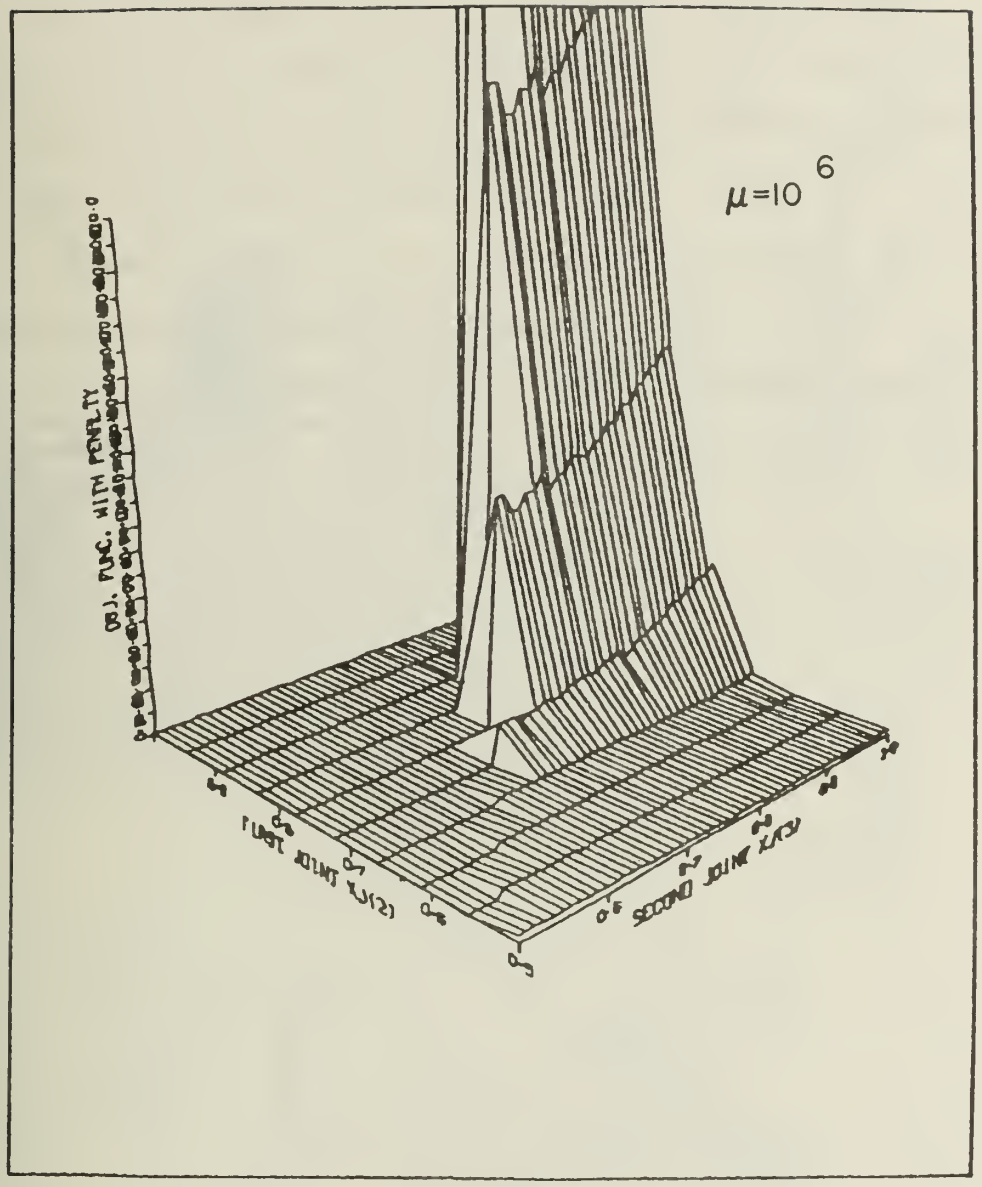


Figure 4.12 Influence of μ; μ = 1,000,000

Therefore X_2 and X_3 should move to the right, that is towards the 1. point end of the interval.

The "penalty wall" will become active for higher values of X_2 and X_3. Figures 4.13 through 4.16 confirm and illustrate the effects due to an increase in R_0 from R_0 = 200 to R_0 = 1K ROM words. The "wall" moves further down away from the X_2Z plane and the minimum is displaced towards (1.,1 .). When R_0 = 1024 the number of ROM words made available is large enough to allow a polynomial of degree 1 evaluation over the whole range ($X_2 = X_3 = 1$).

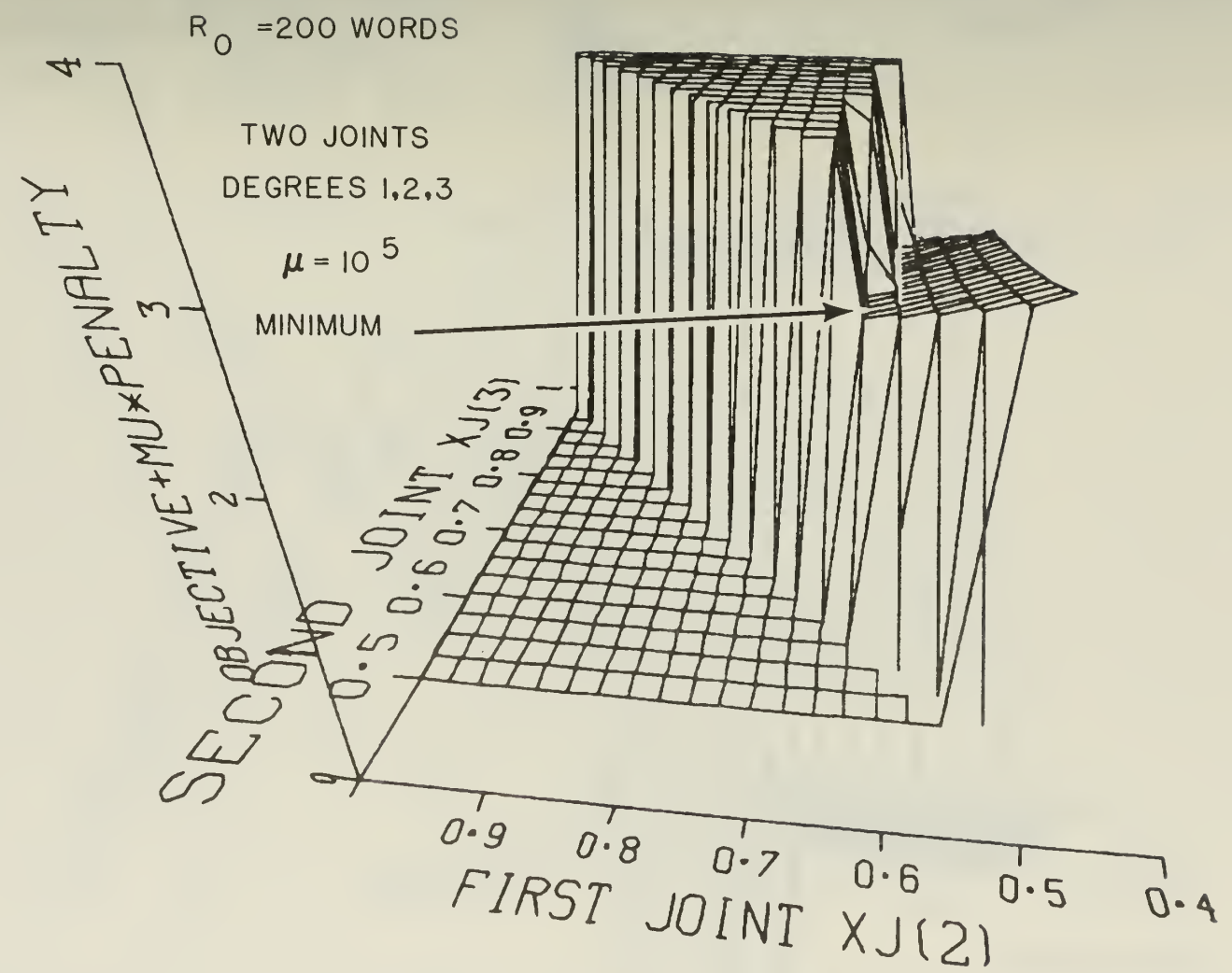


Figure 4.13 Influence of the Maximum Number of ROM Words R_o on the Position of the Minimum; R_o = 200

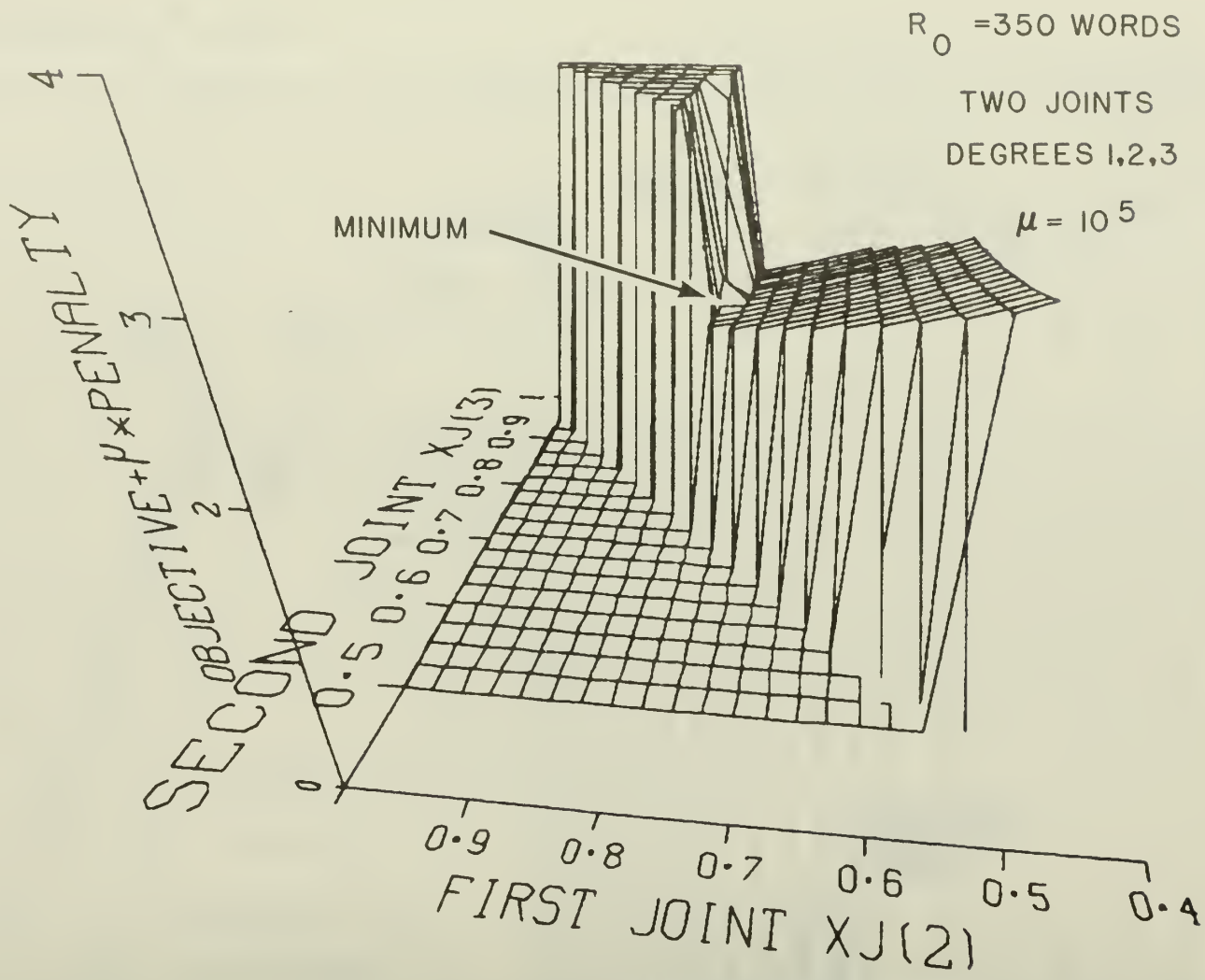


Figure 4.14 Influence of the Maximum Number of ROM Words R_o on the Position of the Minimum; R_o = 350

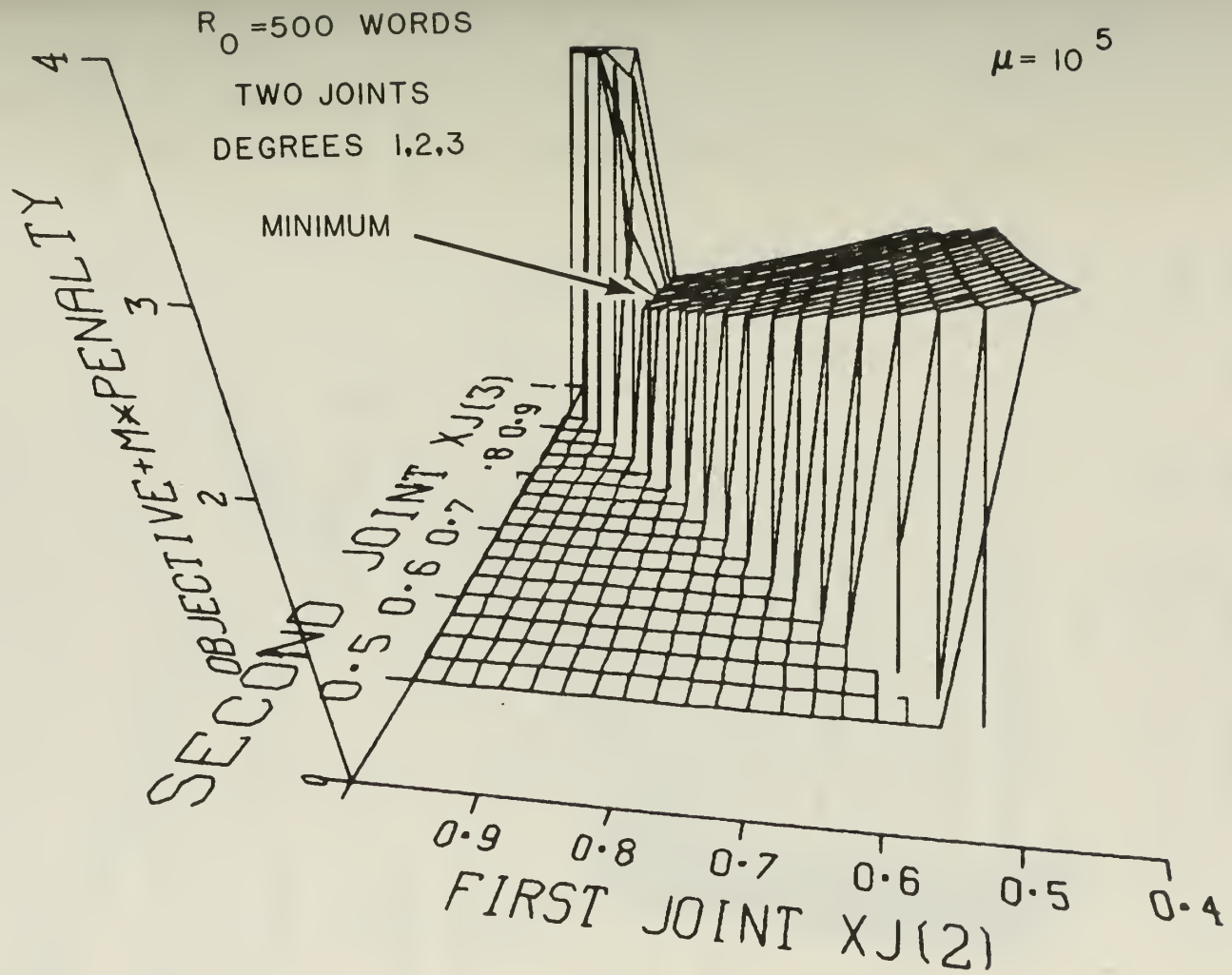


Figure 4.15 Influence of the Maximum Number of ROM Words R_o on the Position of the Minimum; R_o = 500

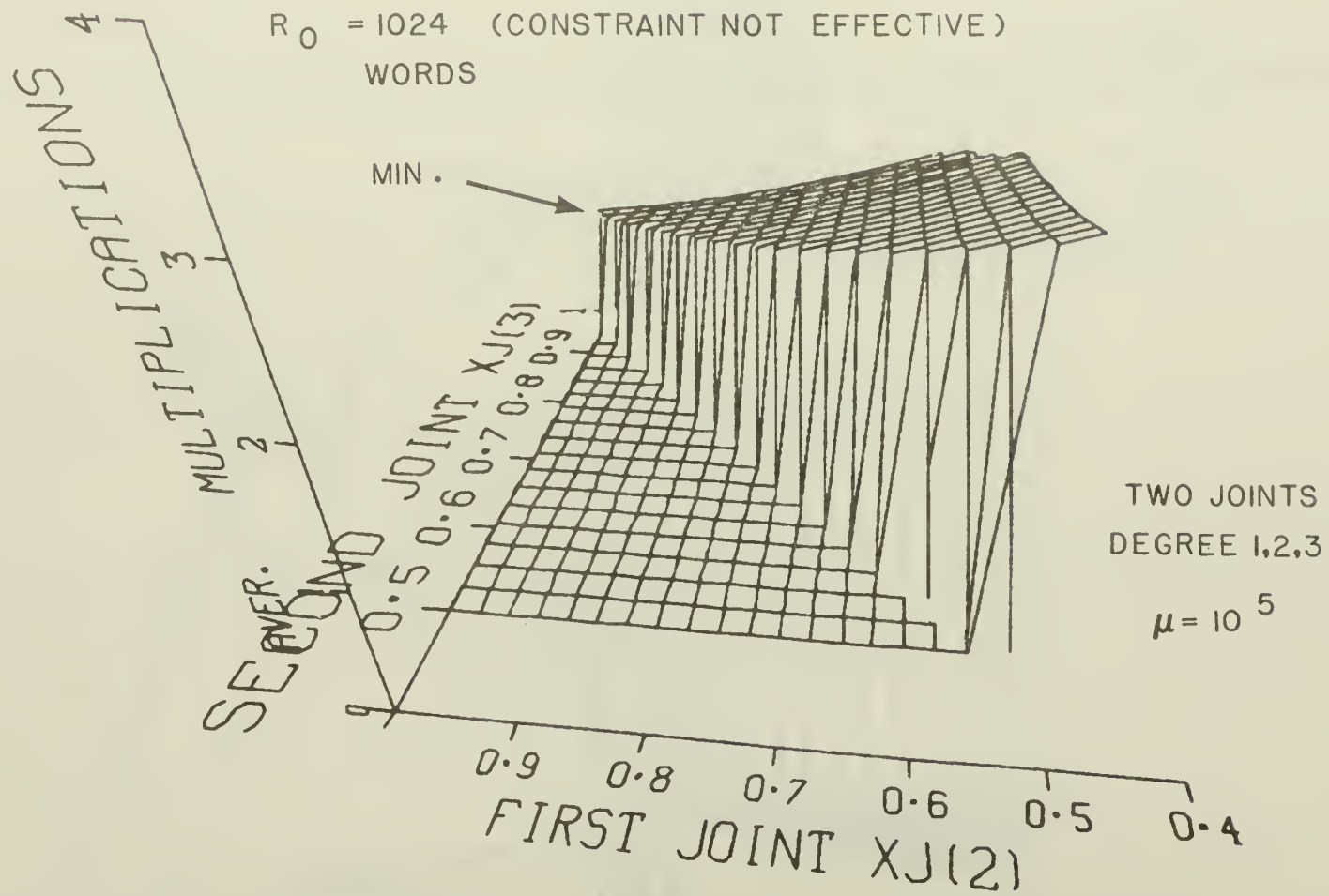


Figure 4.16 Influence of the Maximum Number of ROM Words R_o on the Position of the Minimum; R_o = 1024

5. RESULTS AND EXAMPLES

5.1 Average Number of Multiplications vs. Number of ROM Words: Optimal Curves

The optimization program was run for various values of the maximum number of available Read Only Memory words R_0. The cases chosen are listed in Table 5.1.

Each point on these optimum curves was obtained as a solution of the non-linear programming problem as explained in Section 4. The preliminary study of Section 4 made it possible to avoid having to use a grid search for the initialization of the program. Although the function to be minimized is not unimodal, starting with XJ(1) = XJ(2) = .5 insures the convergence to the minimum. This is due to the fact that the degrees of the polynomials increase from left to right. The number of points computed was limited by computer time and cost considerations. Solutions were sought having 3 significant digits. The IMSL minimization routine (ZXMIN) allows this choice. Depending on the specific case, the range of R_0 values is [25, 650] and the value of ε_o is 2^{-24}.

Type	Penalty Constant μ	Function f(x)	Number of Joints	Polynomial Degrees Used	Results presented in
VIVD	10^5	1/2	1	1,2	Fig. 5.2
VIVD	10^5	1/x	1	1,3	Fig. 5.3
VIVD	10^5	1/x	1	1,4	Fig. 5.4
VIVD	10^5	1/x	1	1,5	Fig. 5.5
VIVD	10^5	1/x	1	2,3	Fig. 5.6
VIVD	10^5	1/x	1	2,4	Fig. 5.7
VIVD	10^5	1/x	1	2,5	Fig. 5.8
VIVD	10^5	1/x	1	3,4	Fig. 5.9
VIVD	10^5	1/x	1	3,5	Fig. 5.10
VIVD	10^5	1/x	1	4,5	Fig. 5.11
VIVD	10^5	1/x	2	1,2,3	Fig. 5.15
VIVD	10^5	1/x	2	1,2,4	Fig. 5.16
VIVD	10^5	1/x	2	1,2,5	Fig. 5.17
VIVD	10^5	1/x	2	1,3,4	Fig. 5.18
VIVD	10^5	1/x	2	1,3,5	Fig. 5.19
VIVD	10^5	1/x	2	1,4,5	Fig. 5.20
VIVD	10^5	1/x	2	2,3,5	Fig. 5.21
VIVD	10^5	1/x	2	2,4,5	Fig. 5.22

Table 5.1

Conditions Used for Figures 5.2 through 5.22

5.1.1 One Joint Results

Figure 5.1 shows clearly the origin of each point on the optimal curves. In general, when the number of ROM words available increases, the average number of multiplications decreases. The shape of the curve depends on the degrees of the polynomials chosen. In all cases, if the polynomial degrees are m_1, m_2, m_3, where $m_1 < m_2 < m_3$, the average number of multiplications decreases between m_1 and m_3. In some cases, the curve levels off above a given number of ROM words (for example, polynomial degrees 2, 3, Fig. 5.6). This is due to the fact that penalty does not apply and we have a degenerate case where the approximation is done with the lowest polynomial degree over the whole range. For example, in the case of polynomials of degrees 2 and 3 beyond 150 ROM words, the average (and in this case the true) number of multiplications is 2. That is, there are no polynomials of degree 3 used in the range [.5,1] and the two joints are located by the program at the extreme right ($X_2 = X_3 = X_4 = 1$). In this example the point where the 3rd degree polynomials are no longer needed is at approximately $R_0 = 150$.

The one joint results are given in Fig. 5.2 - 5.11. Fig. 5-12 represents a partial ordering between all the one joint implementations. There are two possible cases:

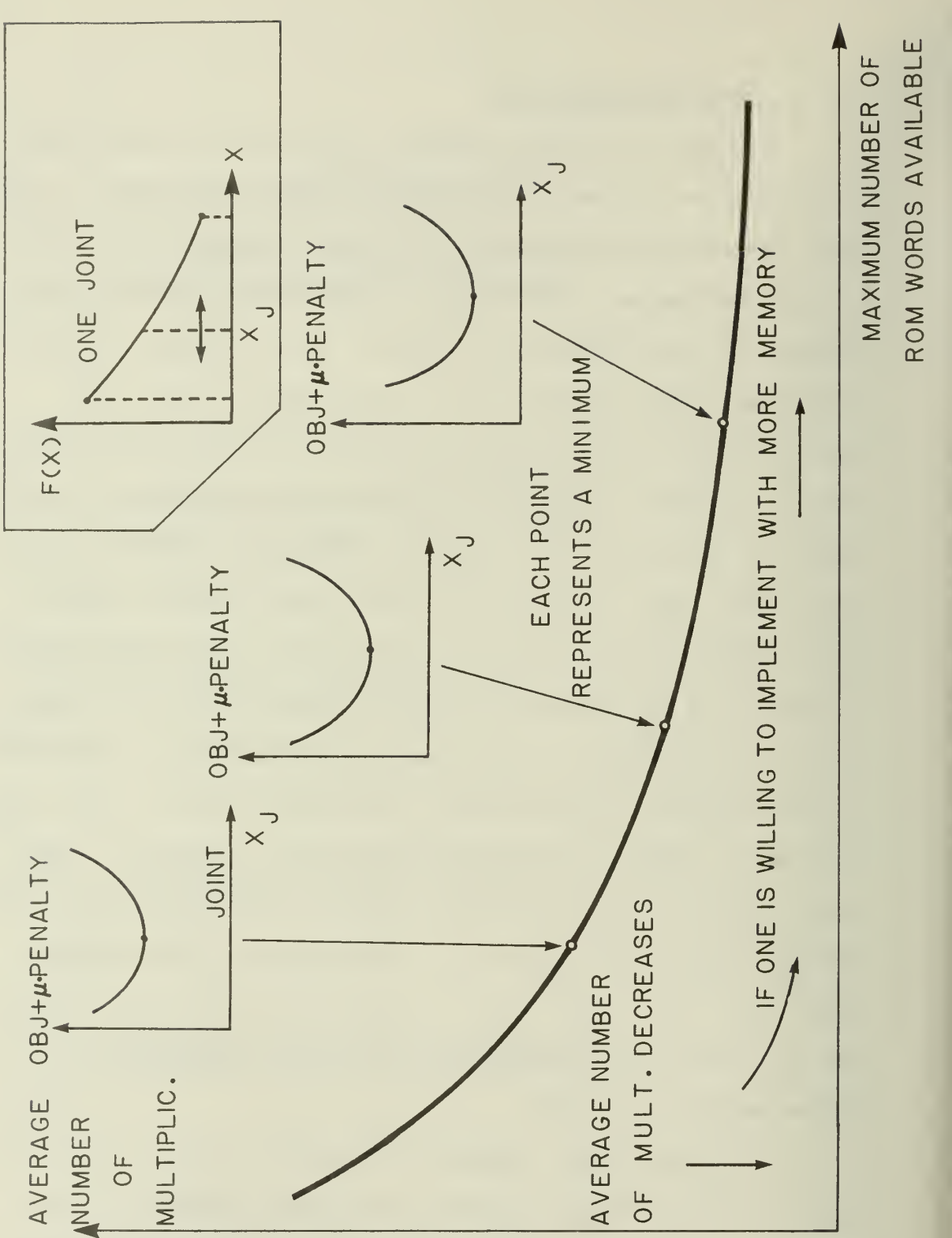


Figure 5.1 Each Point on the Curve Represents a Minimum

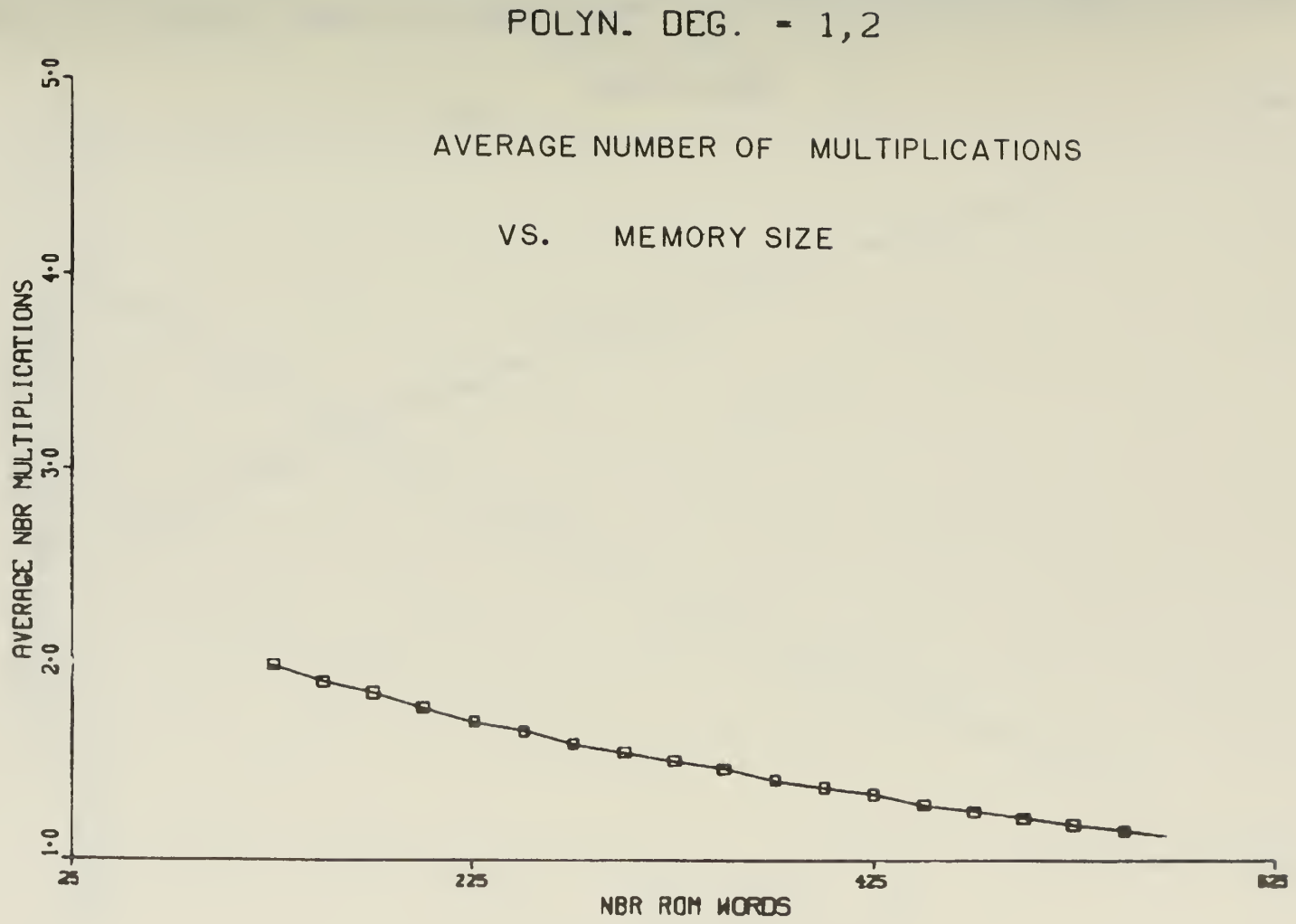


Figure 5.2 Optimal Curve: Average Number of Multiplications vs Cost in Number of ROM Words; VIVD Case; One Joint; Polyn. Deg. = 1,2

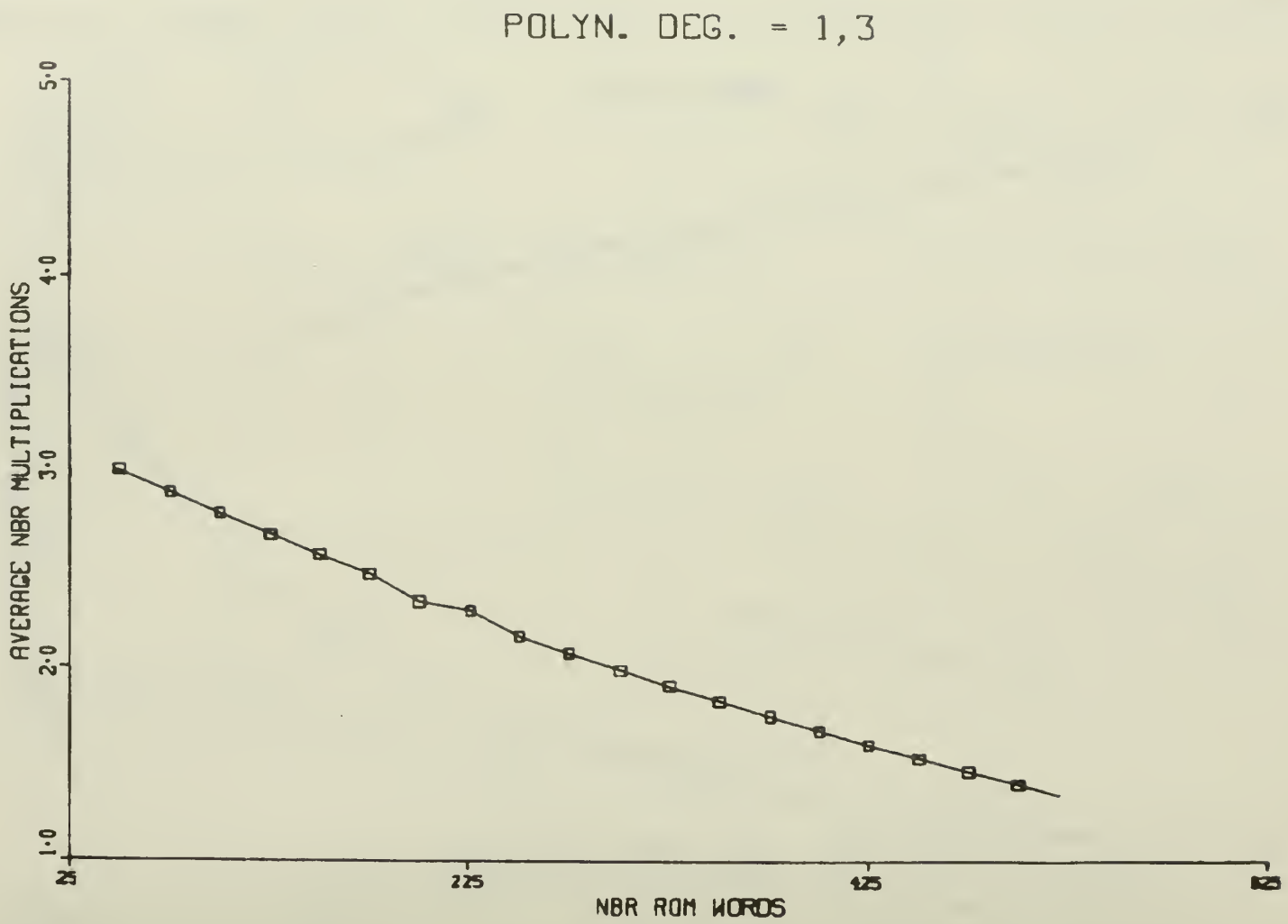


Figure 5.3 Optimal Curve: Average Number of Multiplications vs Cost in Number of ROM Words; VIVD Case; One Joint; Polyn. Deg. = 1,3

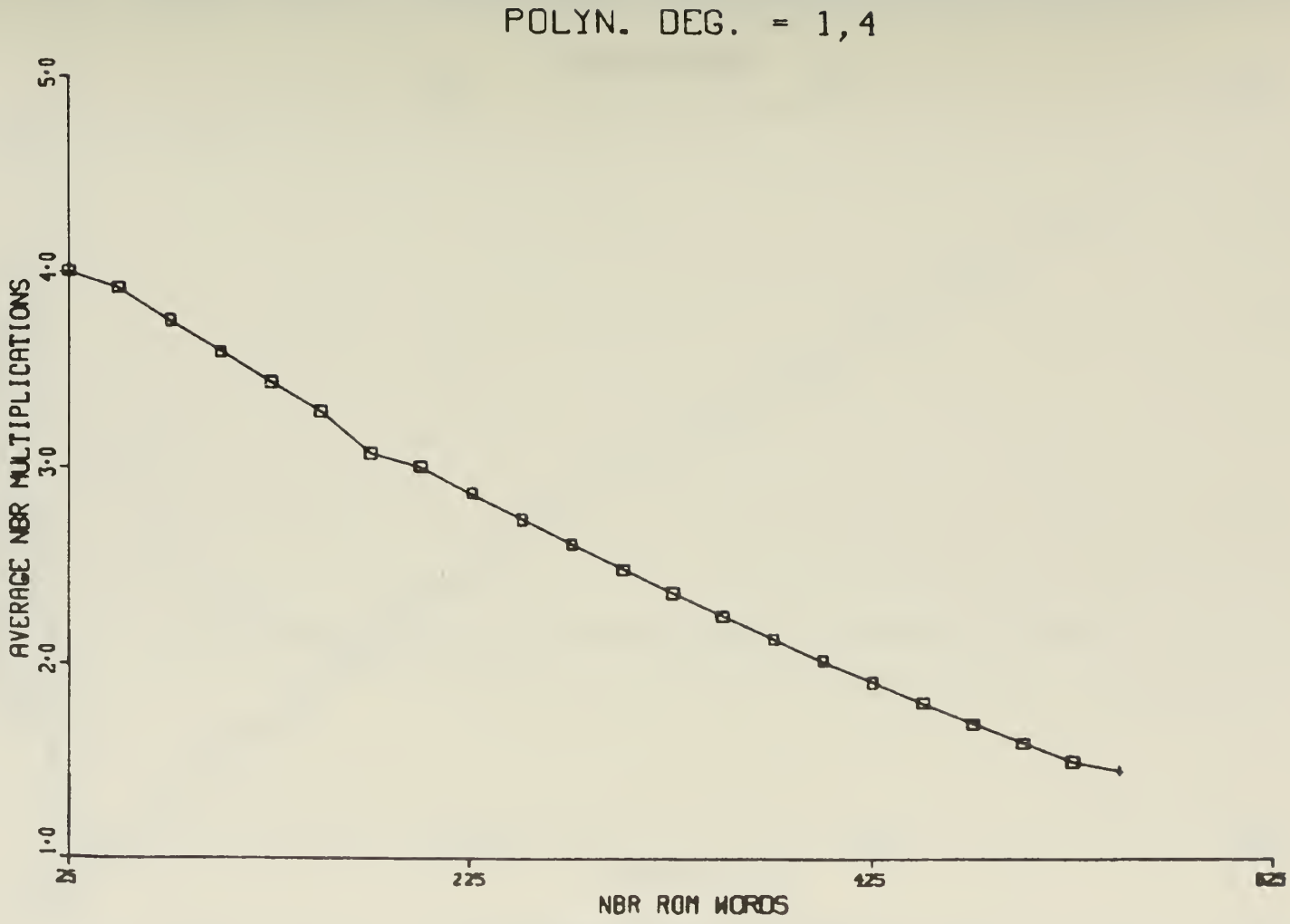


Figure 5.4 Optimal Curve: Average Number of Multiplications vs Cost in Number of ROM Words; VIVD Case; One Joint; Polyn. Deg. = 1,4

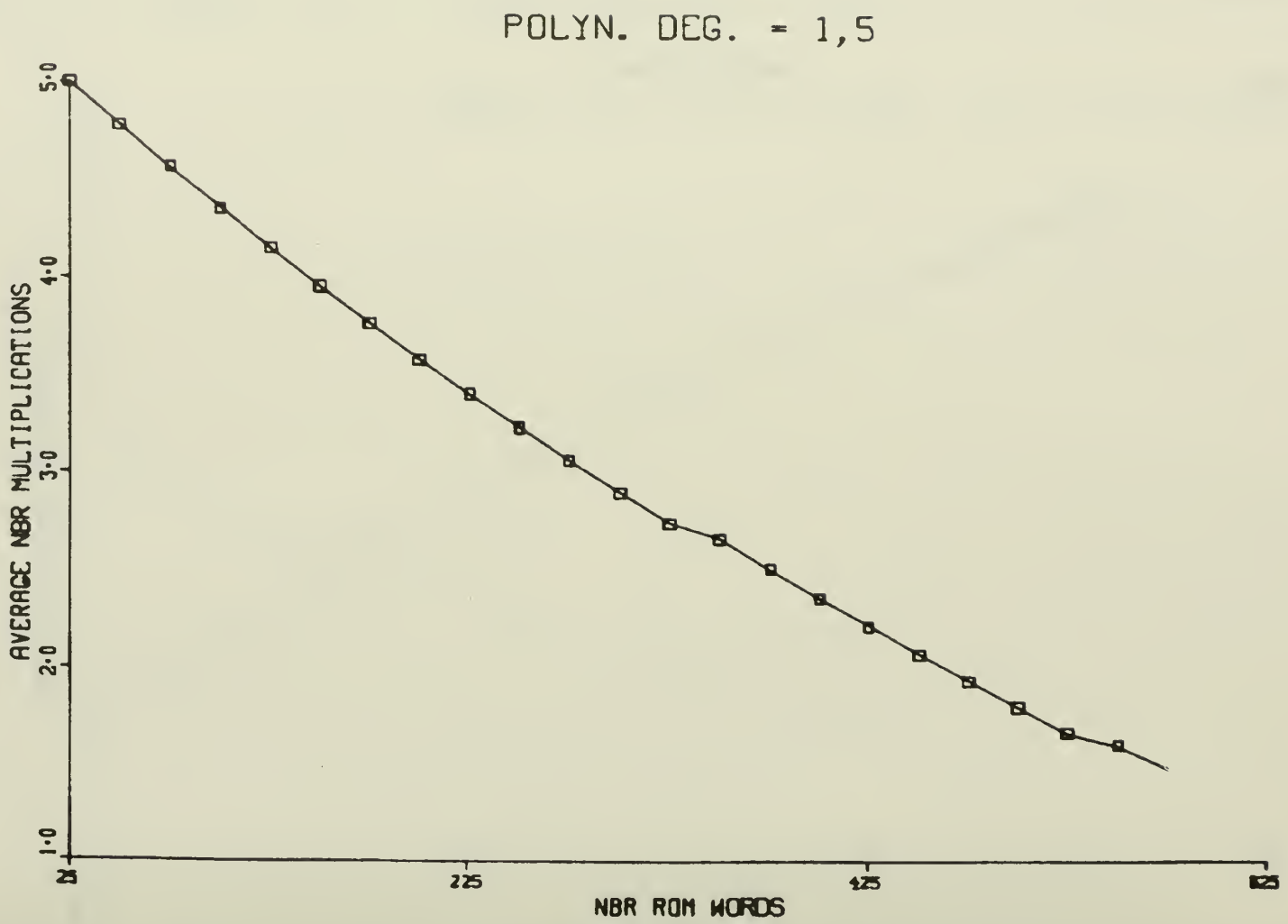


Figure 5.5 Optimal Curve: Average Number of Multiplications vs Cost in Number of ROM Words; VIVD Case; One Joint; Polyn. Deg. = 1,5

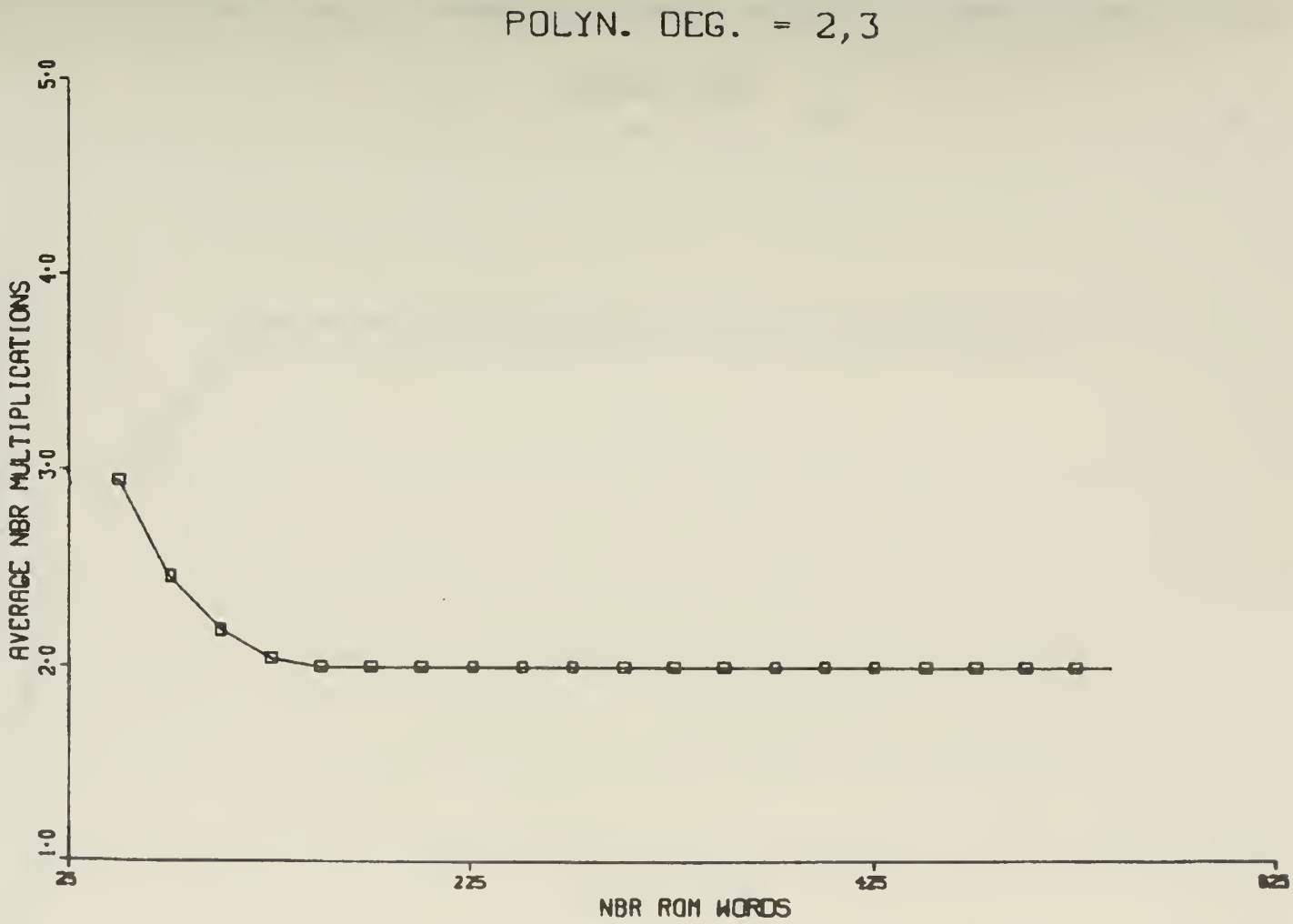


Figure 5.6 Optimal Curve: Average Number of Multiplications vs Cost in Number of ROM Words; VIVD Case; One Joint; Polyn. Deg. = 2,3

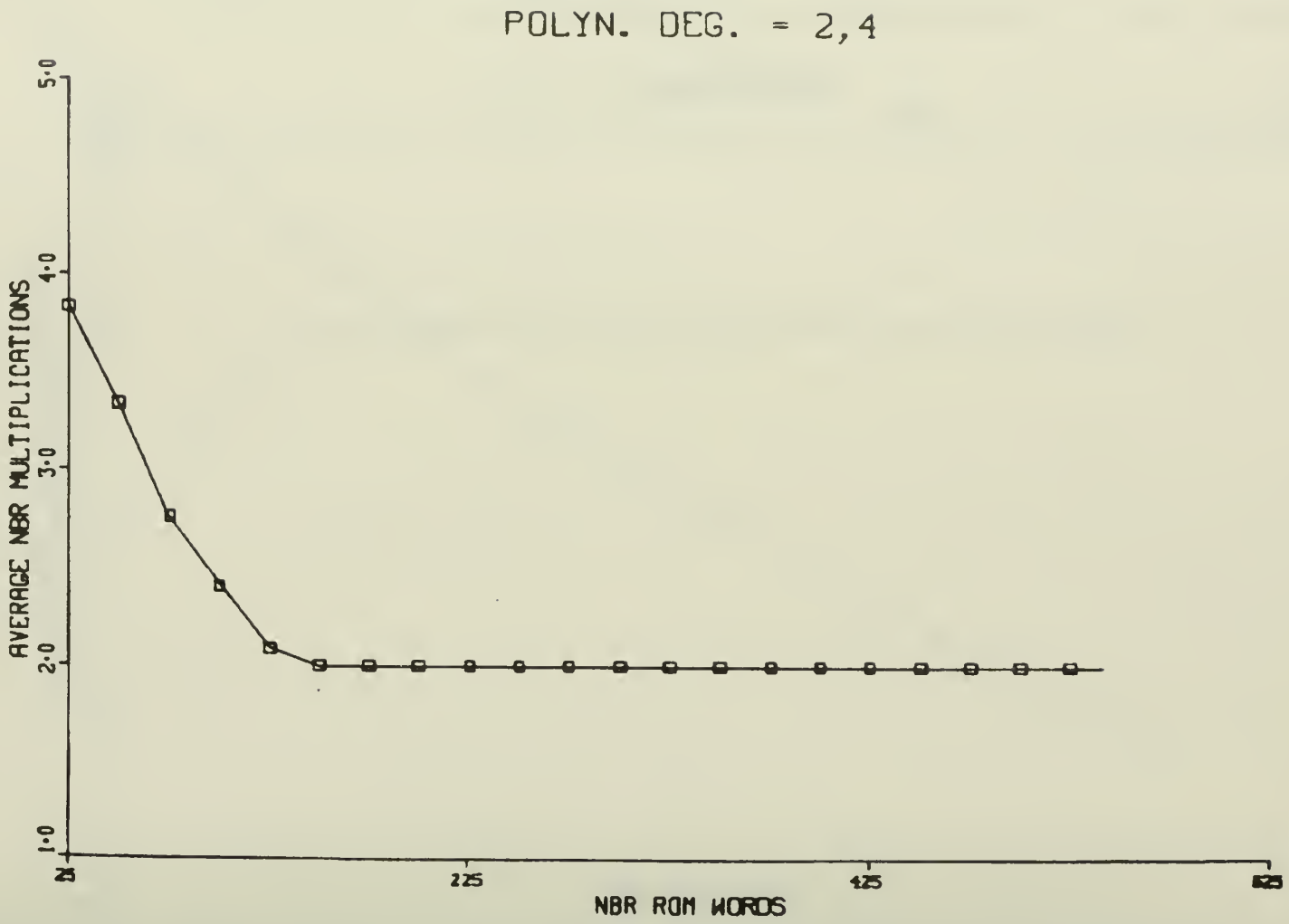


Figure 5.7 Optimal Curve: Average Number of Multiplications vs Cost in Number of ROM Words; VIVD Case; One Joint; Polyn. Deg. = 2,4

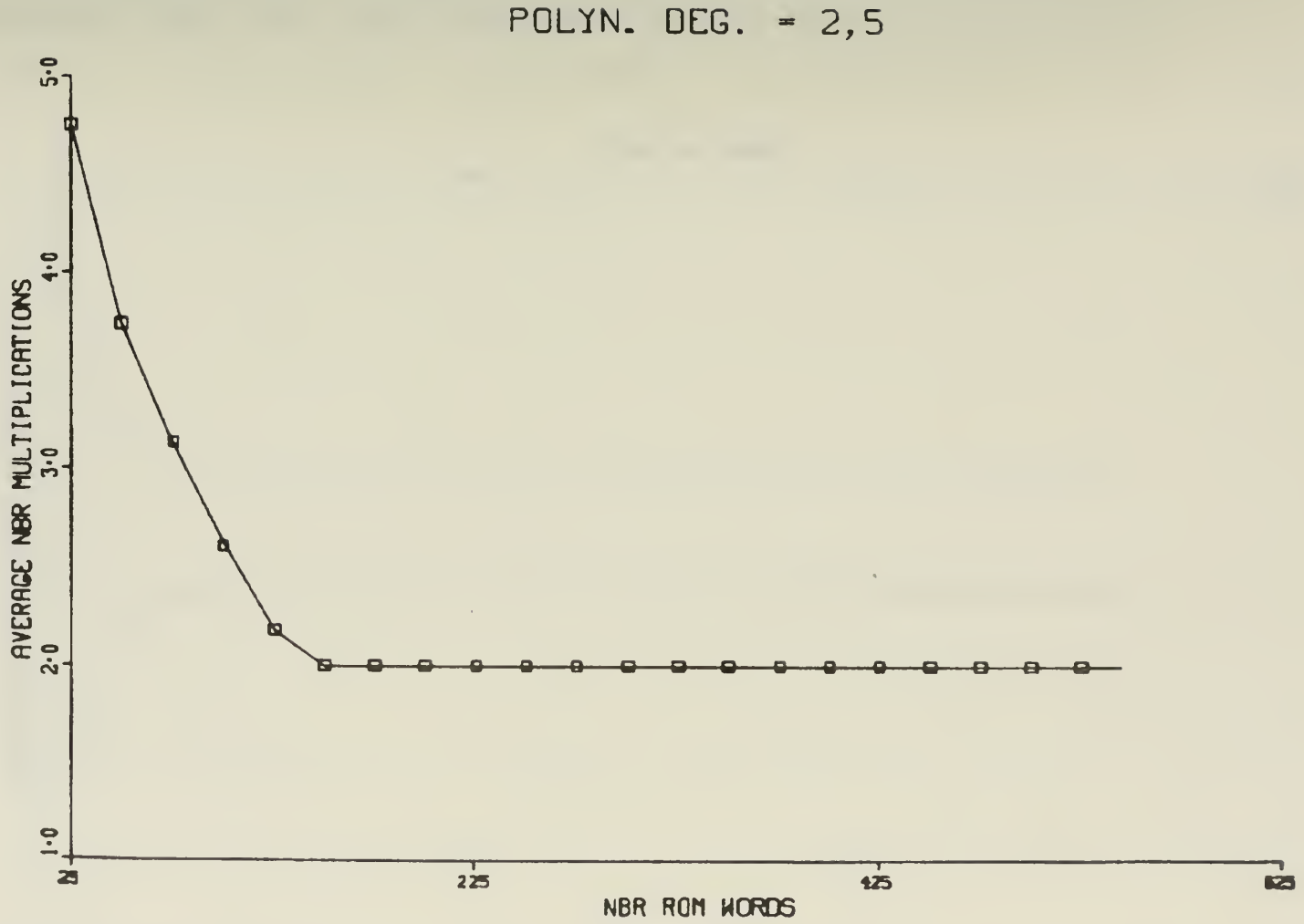


Figure 5.8 Optimal Curve: Average Number of Multiplications vs Cost in Number of ROM Words; VIVD Case; One Joint; Polyn. Deg. = 2,5

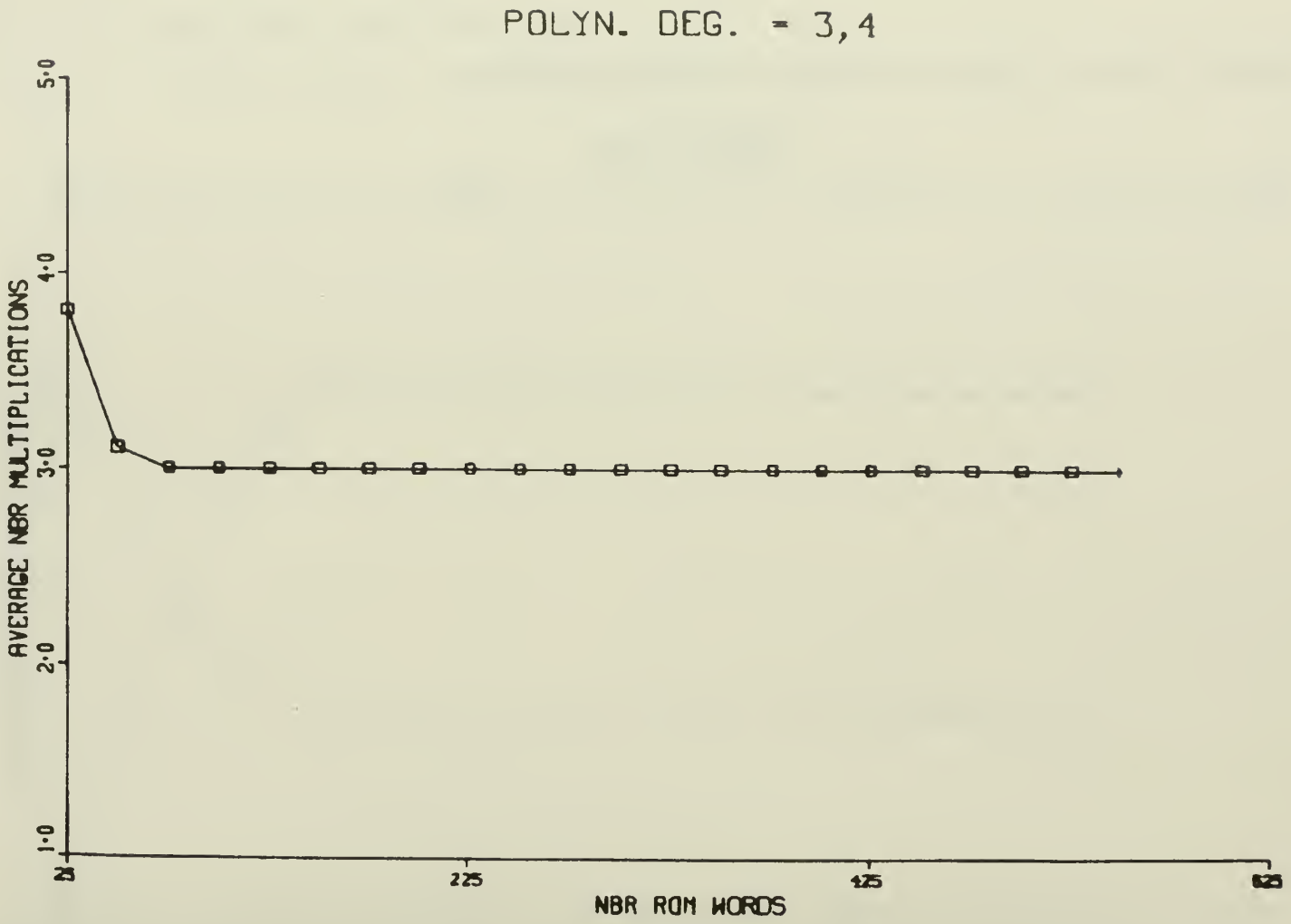


Figure 5.9 Optimal Curve: Average Number of Multiplications vs Cost in Number of ROM Words; VIVD Case; One Joint; Polyn. Deg. = 3,4

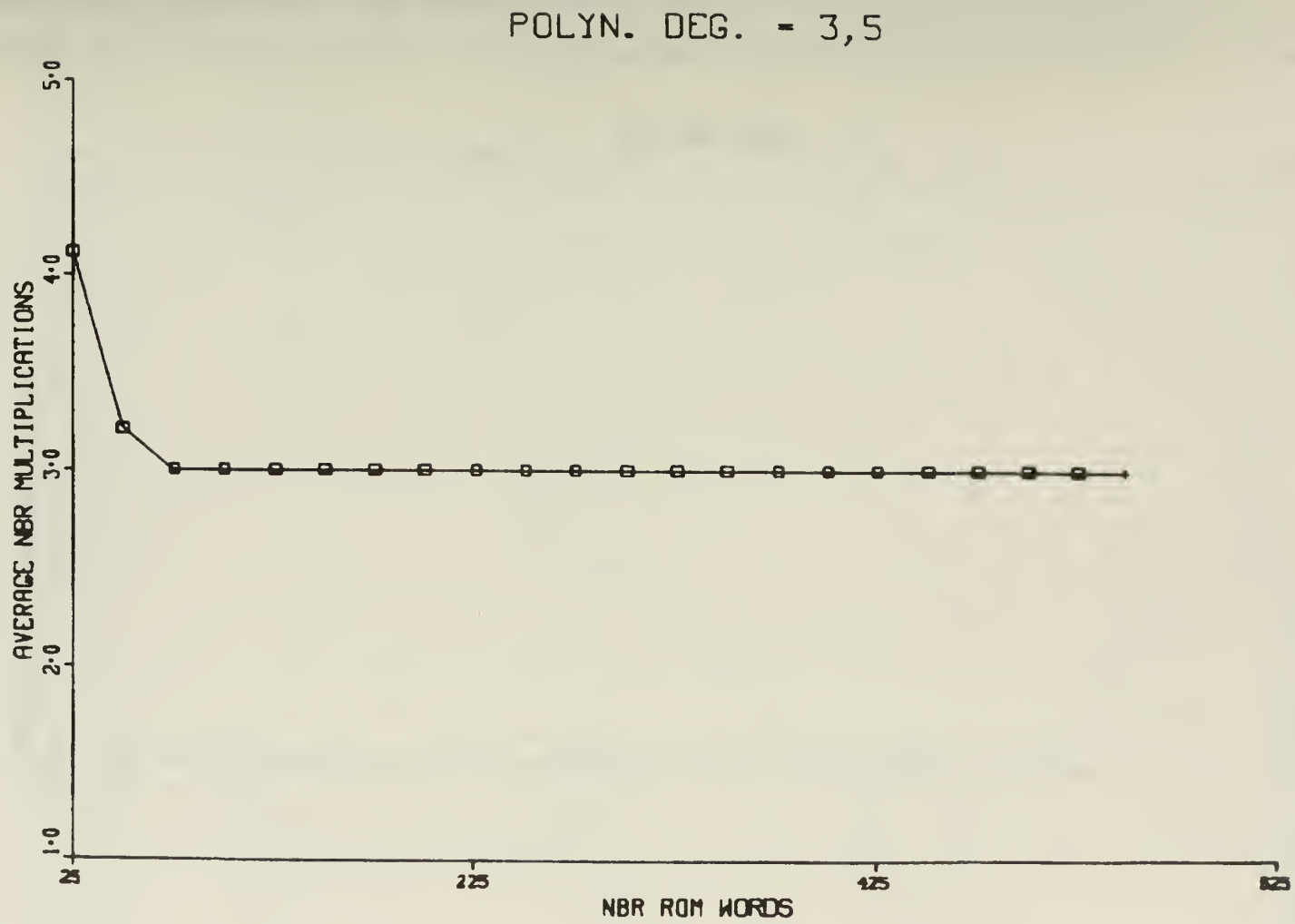


Figure 5.10 Optimal Curve: Average Number of Multiplications vs Cost in Number of ROM Words; VIVD Case; One Joint; Polyn. Deg. = 3,5

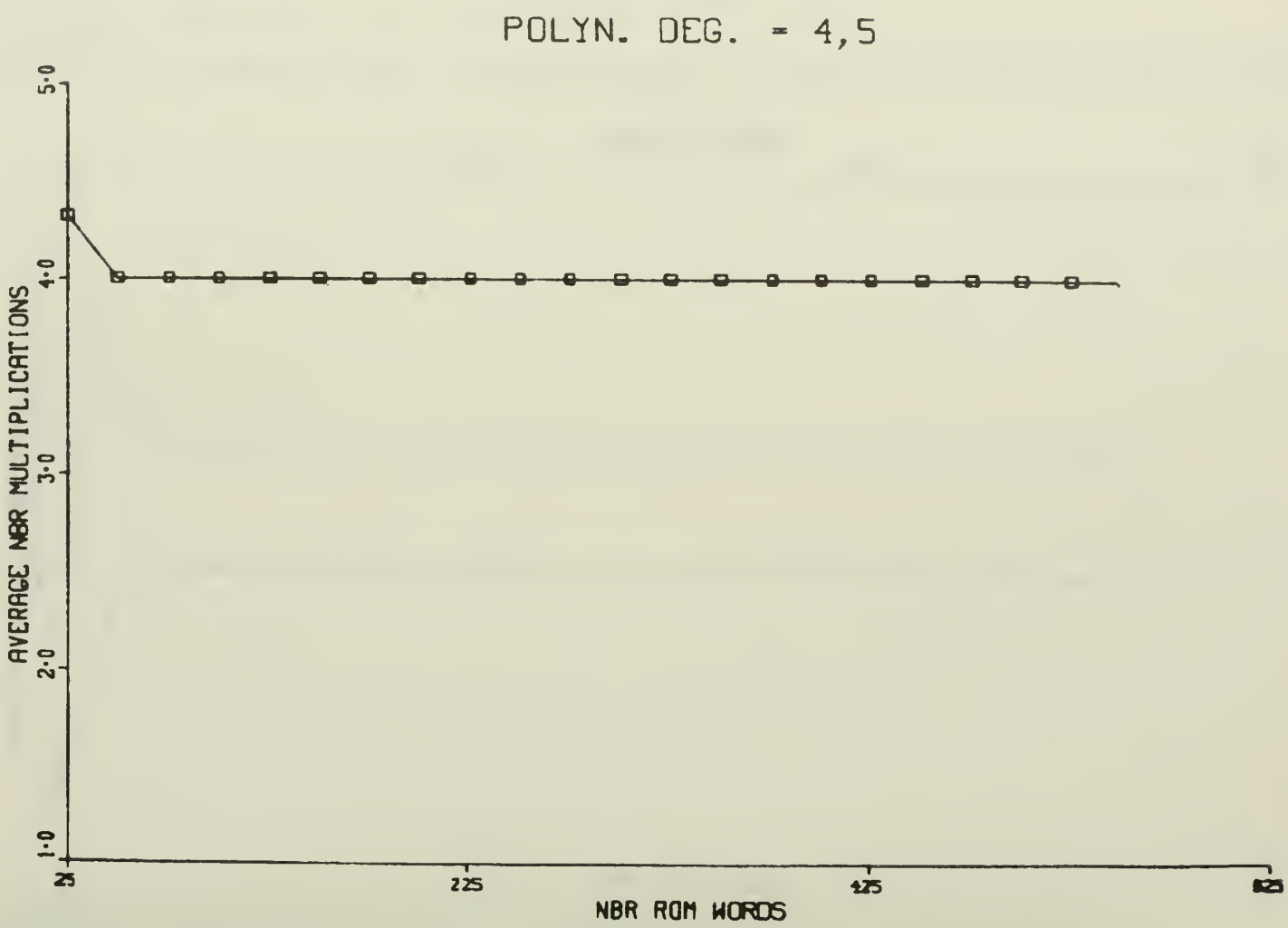


Figure 5.11 Optimal Curve: Average Number of Multiplications vs Cost in Number of ROM Words; VIVD Case; One Joint; Polyn. Deg. = 4,5

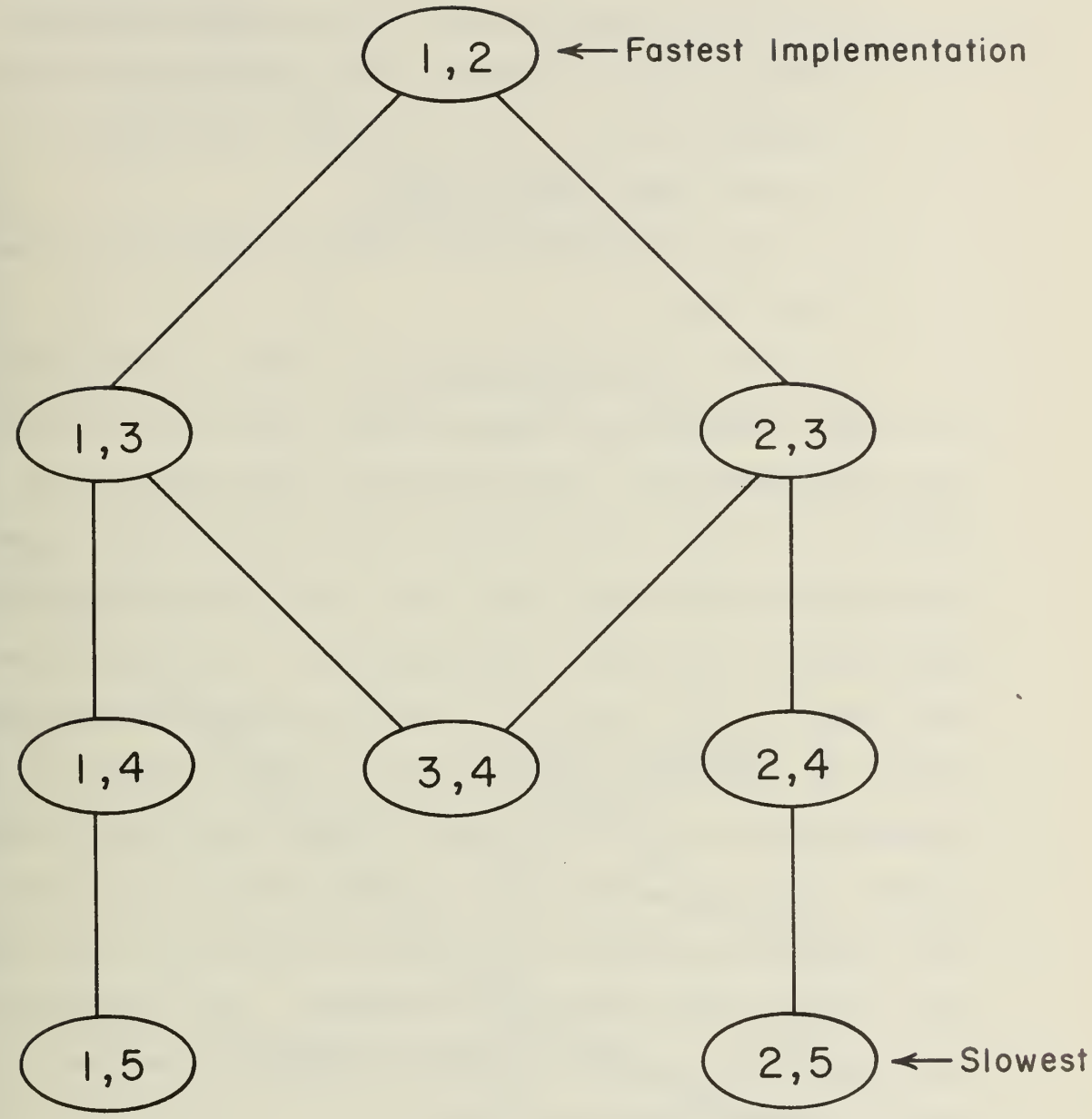


Figure 5.12 Ordering Relationship for Implementations Involving Two Different Polynomials (One Joint)

- either the optimal curve of one implementation is always under the optimal curve for another implementation (e.g.: 1.2 is always faster than 1.3 for the same amount of ROM).
- or the curves cross at a certain value of R_0 (see Fig. 5.13).

In Fig. 5.12, implementations that are comparable are linked together. If they are not (such as 1.3 and 2.3), they cross somewhere. This graph has the properties of a lattice. In Fig. 5.13 where two different optimal curves are plotted on the same graph, the point E corresponds to statistically equivalent implementations. For any point such that $R_0 > R_{0E}$, curve B is obviously more advantageous because it leads to a faster evaluation of the function for the same cost in terms of number of Read Only Memory words. However, this comparison should be weighted by the fact that curve B may involve more joints than curve A and therefore increase the complexity of control. For implementations involving the same number of polynomial degrees, clearly curve B should be chosen.

For any point on the R_0 axis such that $R_0 > R_{0E}$, curve B no longer represents the optimum choice and it is obvious that for the same amount of ROM, curve A now yields a faster implementation.

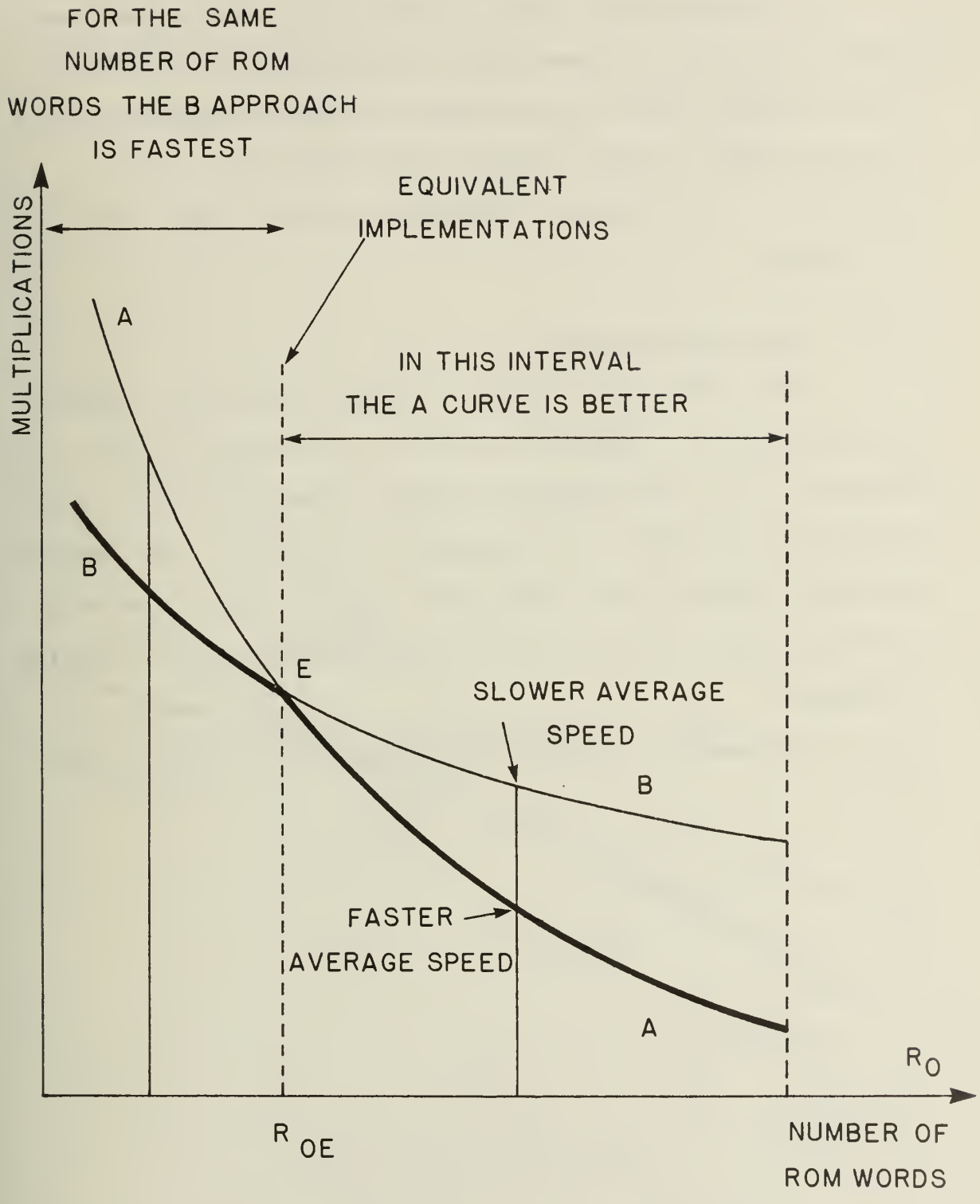


Figure 5.13 The Designer's Choice Among Two Possible Implementations

This situation can, of course, be generalized when there is a choice between many implementations. In Fig. 5.14, the dark heavy curve should be the optimal realization when one is faced between implementations A, B, C, D, etc. . . . . Control complexity may well be a factor, of course.

5.1.2 Two Joint Results

The case where three different polynomial degrees are available for implementation is illustrated in Fig. 5.15 through 5.22. The comparison between these different implementations is shown in the graph of Fig. 5.23. The remarks made for the one joint case apply equally when there is a choice between different possible implementations involving 3 degrees.

An overall ordering relationship is shown in Fig. 5.24.

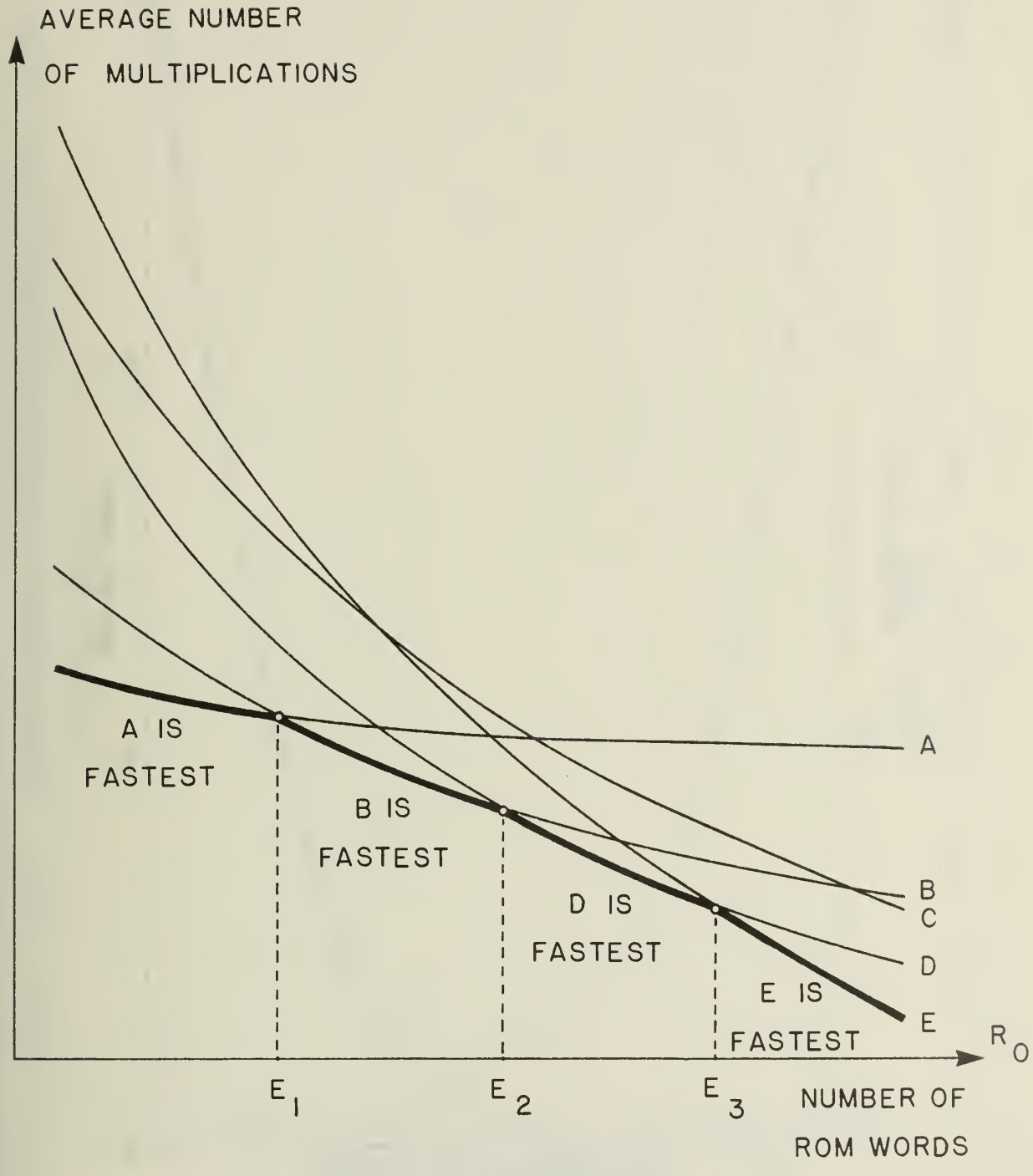


Figure 5.14 The Designer's Choice Among More Than Two Possible Implementations

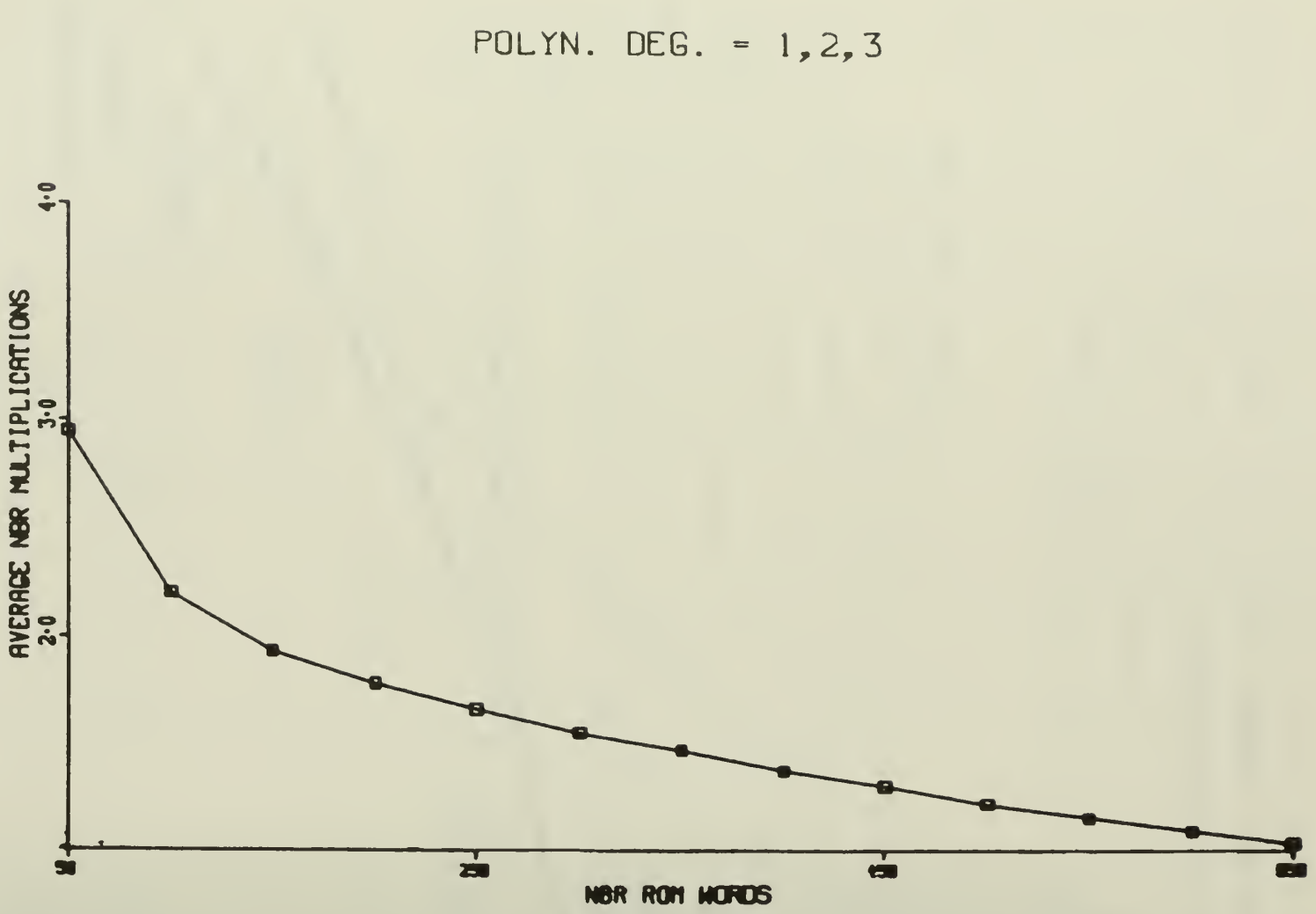


Figure 5.15 Optimal Curve: Average Number of Multiplications vs Cost in Number of ROM Words; VIVD Case; Two Joints; Polyn. Deg. = 1,2,3

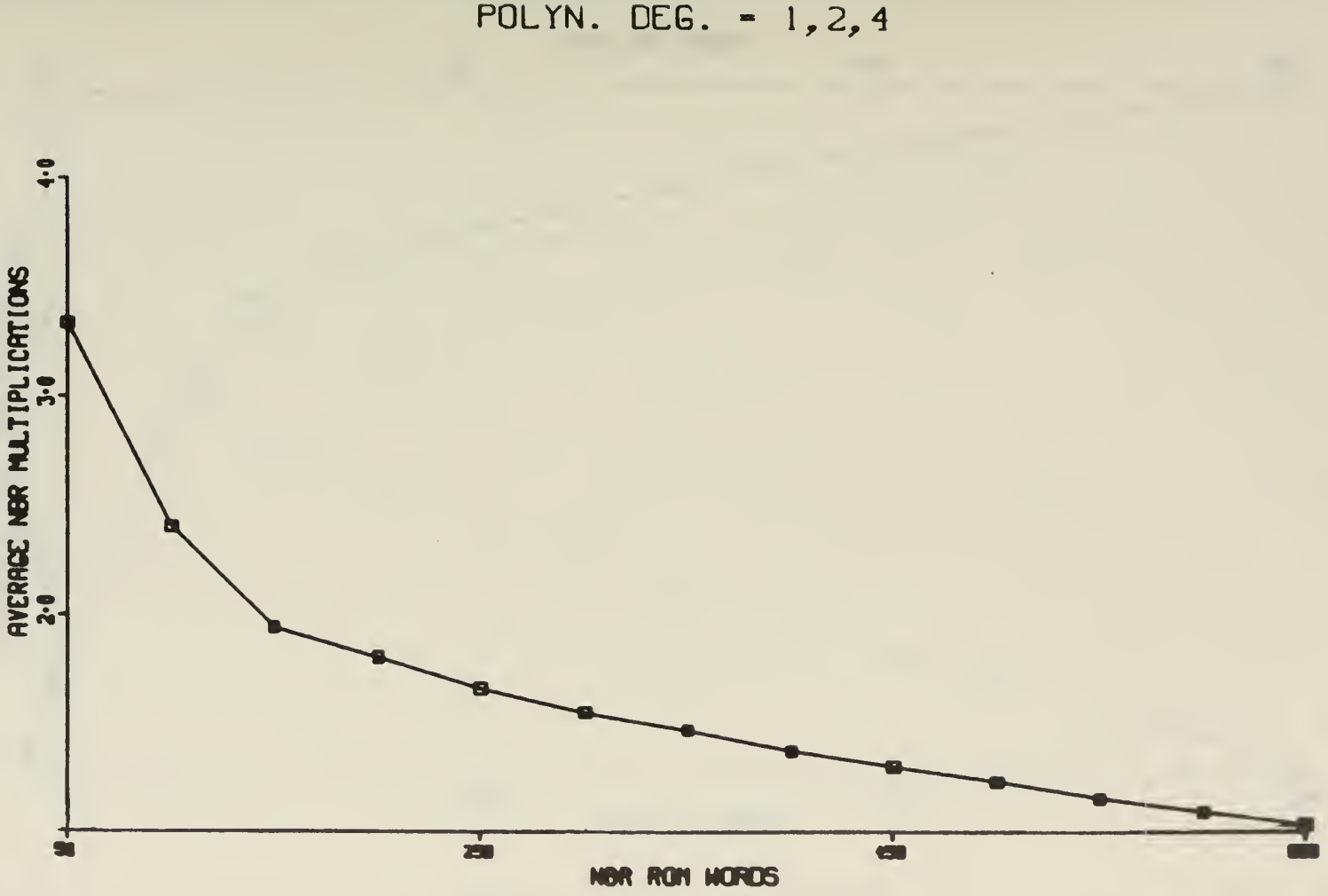


Figure 5.16 Optimal Curve: Average Number of Multiplications vs Cost in Number of ROM Words; VIVD Case; Two Joints; Polyn. Deg. = 1,2,4

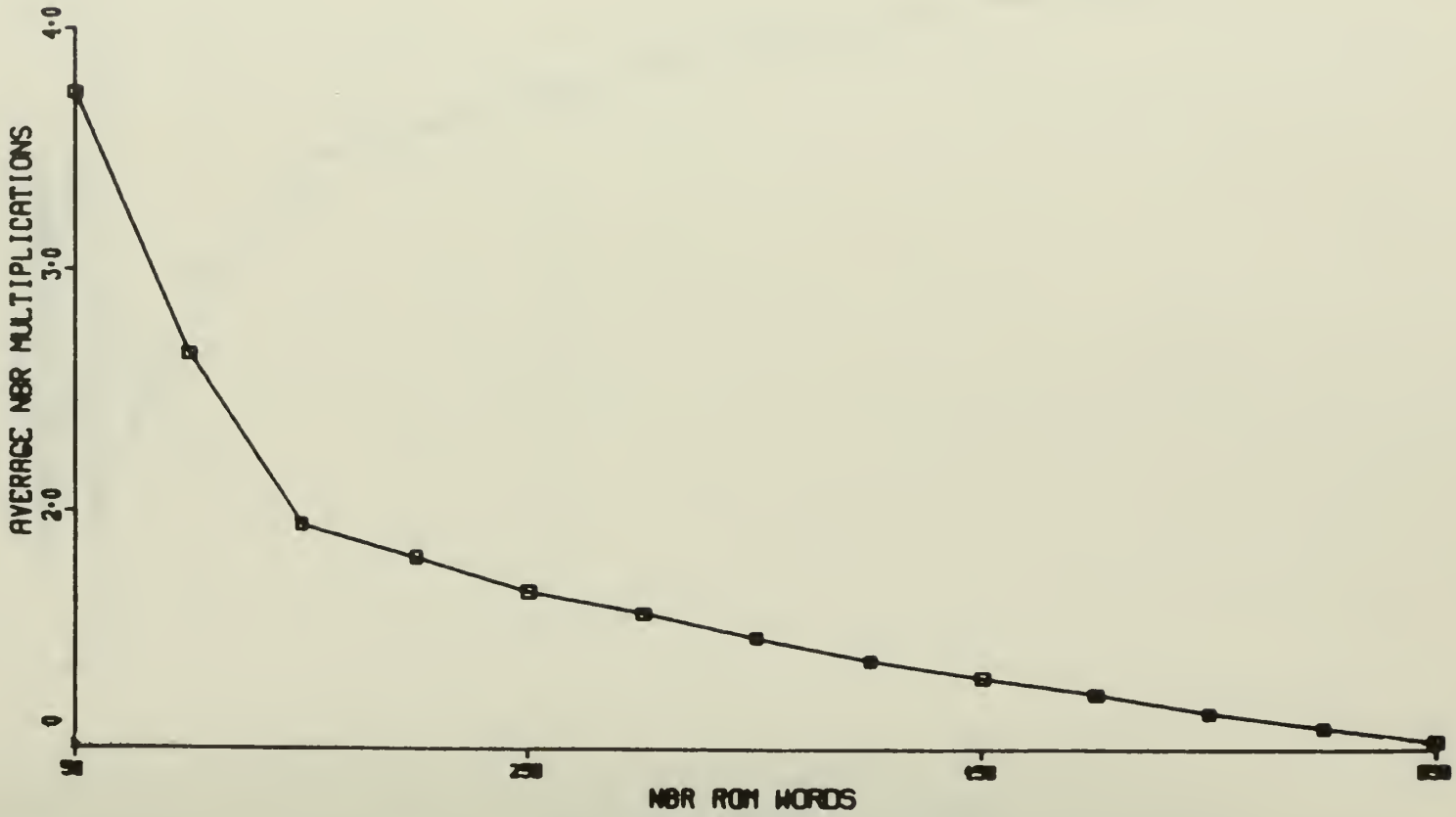


Figure 5.17 Optimal Curve: Average Number of Multiplications vs Cost in Number of ROM Words; VIVD Case; Two Joints; Polyn. Deg. = 1,2,5

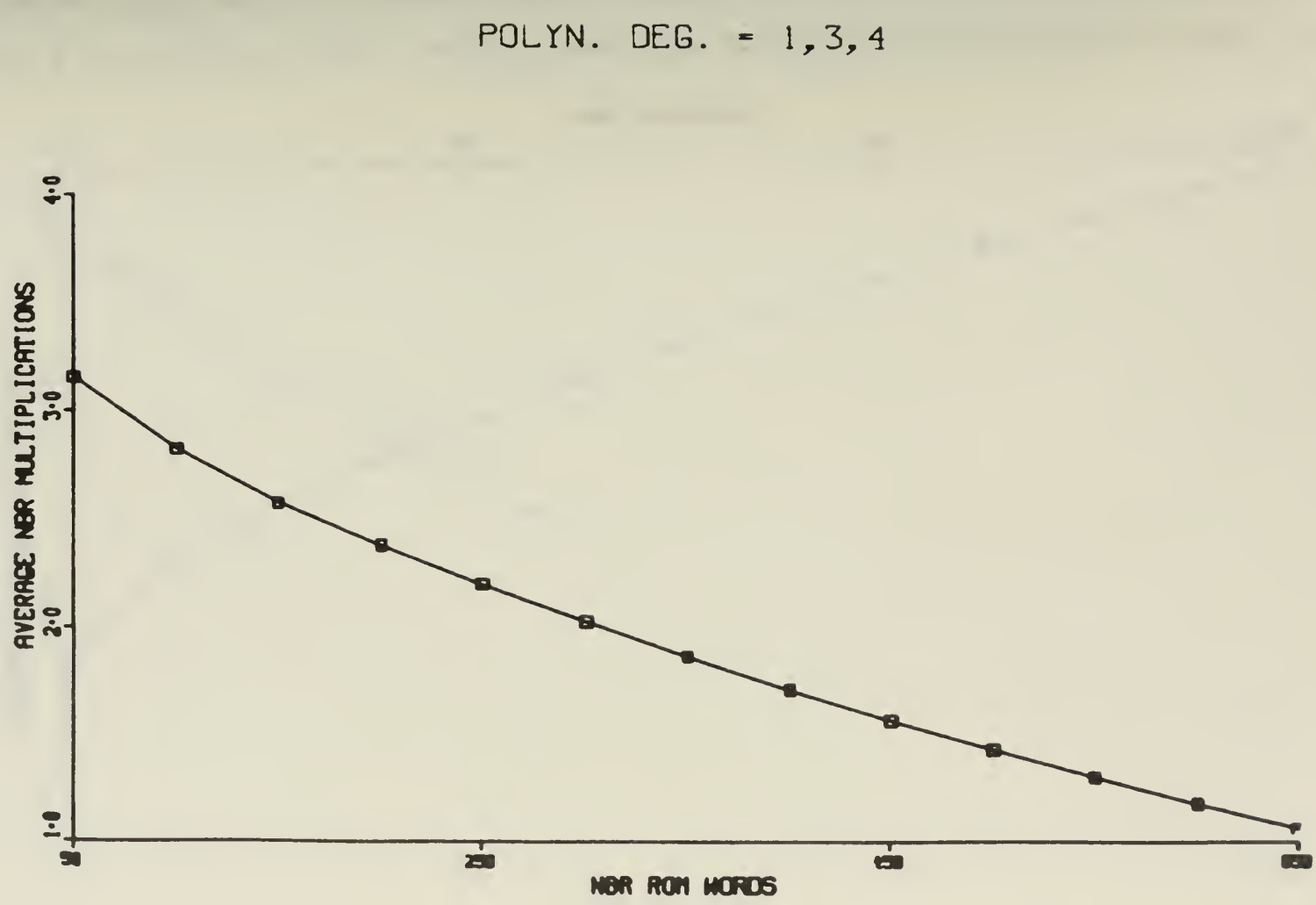


Figure 5.18 Optimal Curve: Average Number of Multiplications vs Cost in Number of ROM Words; VIVD Case; Two Joints; Polyn. Deg. = 1,3,4

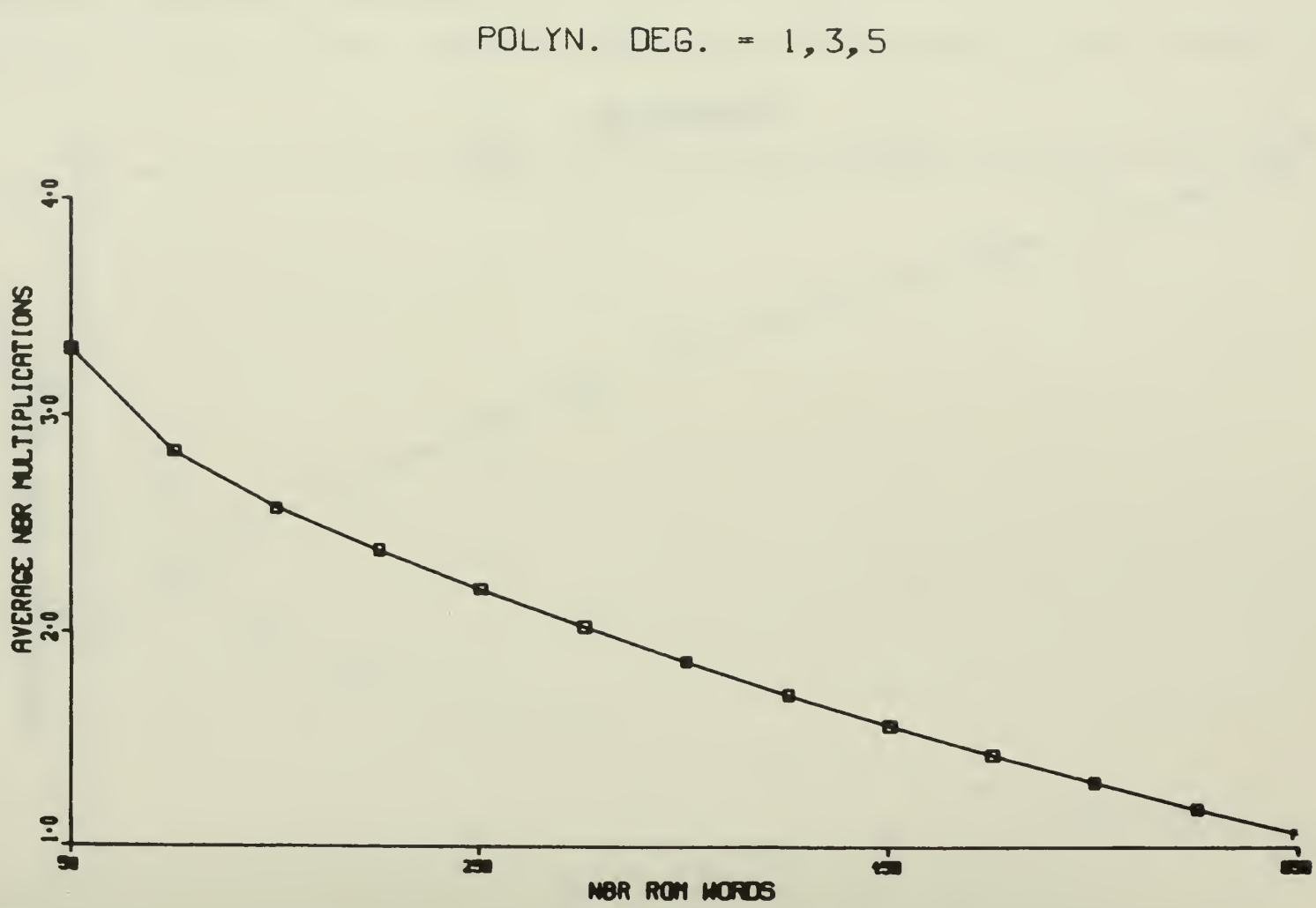


Figure 5.19 Optimal Curve: Average Number of Multiplications vs Cost in Number of ROM Words; VIVD Case; Two Joints; Polyn. Deg. = 1,3,5

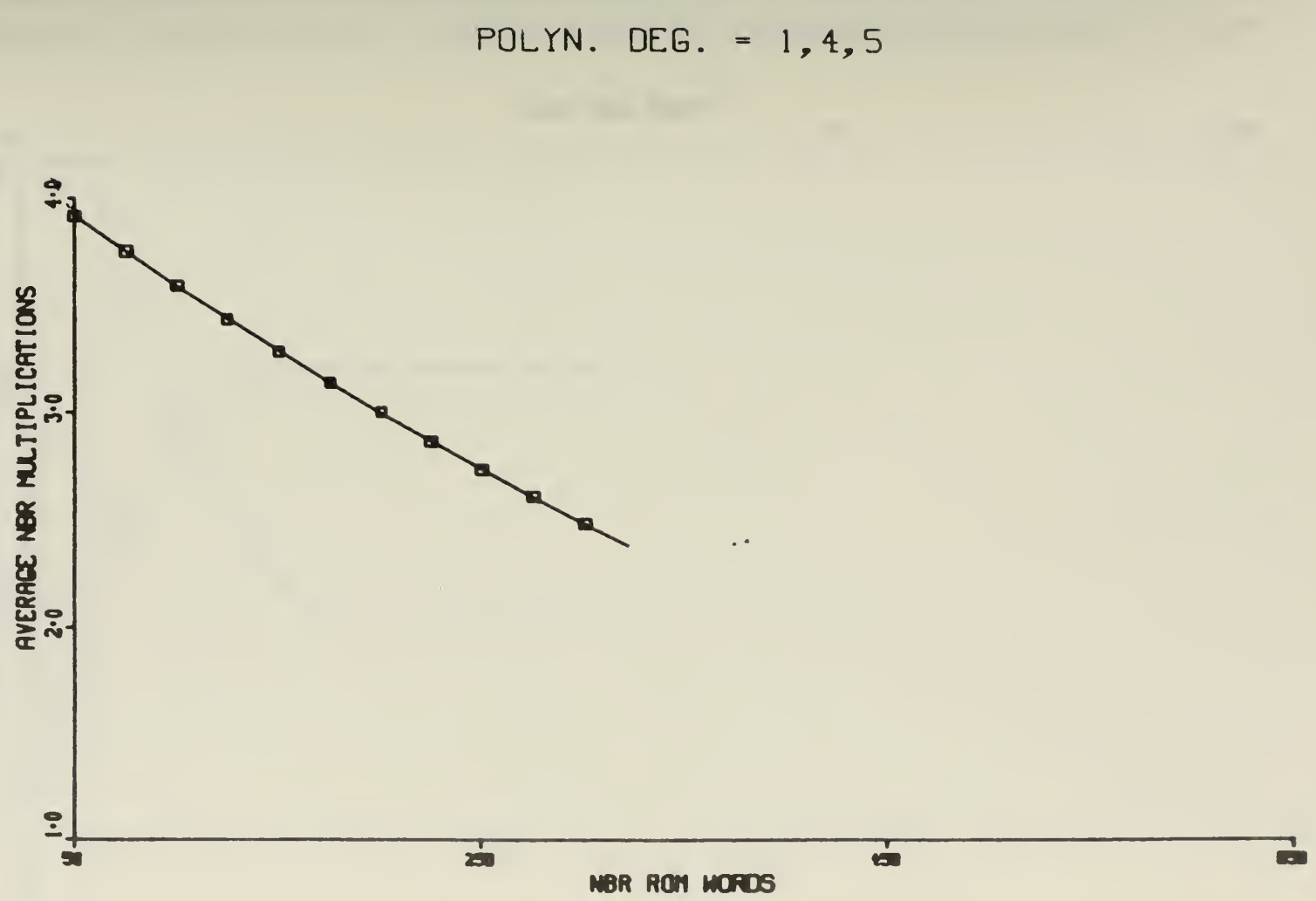


Figure 5.20 Optimal Curve: Average Number of Multiplications vs Cost in Number of ROM Words; VIVD Case; Two Joints; Polyn. Deg. = 1,4,5

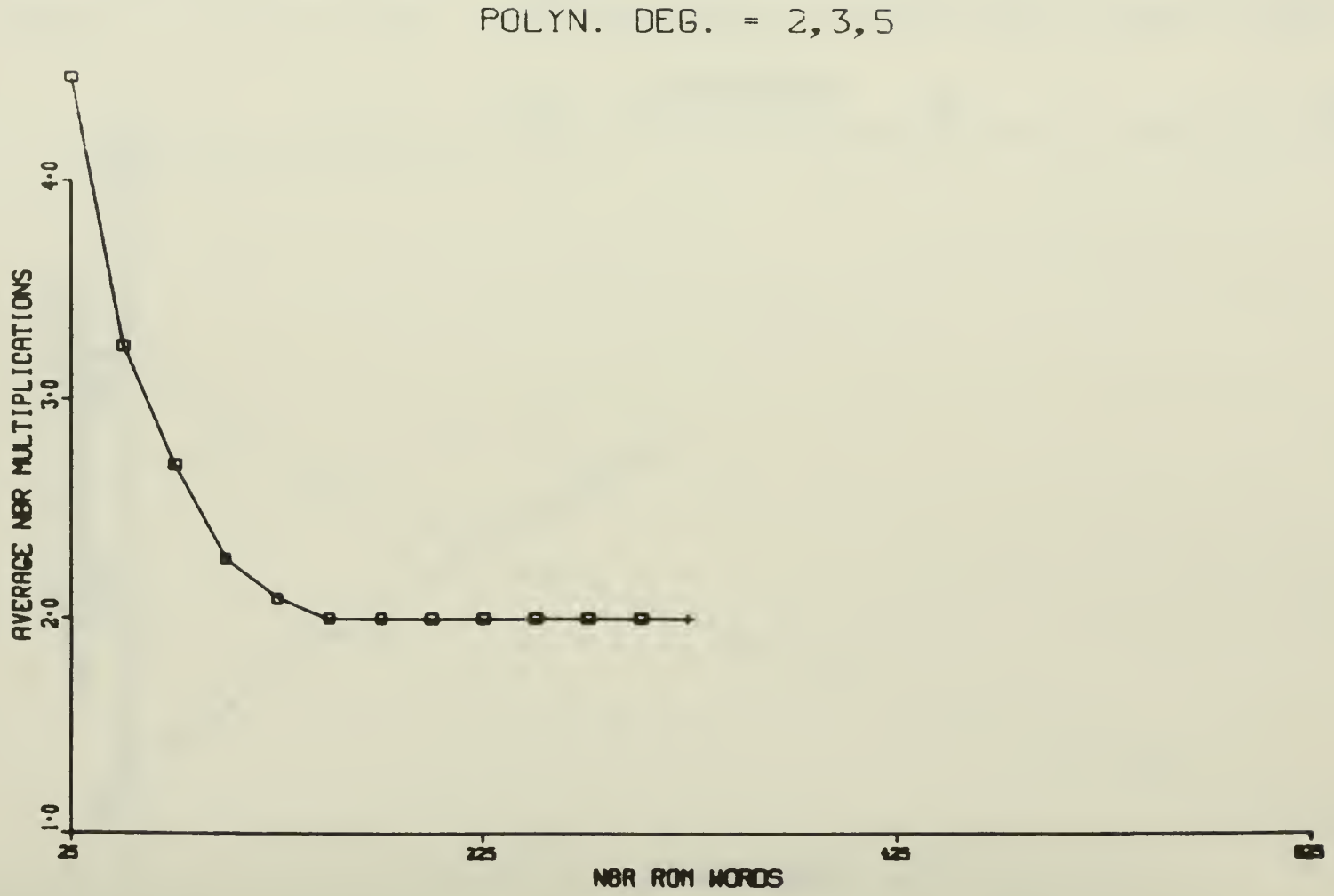


Figure 5.21 Optimal Curve: Average Number of Multiplications vs Cost in Number of ROM Words; VIVD Case; Two Joints; Polyn. Deg. = 2,3,5

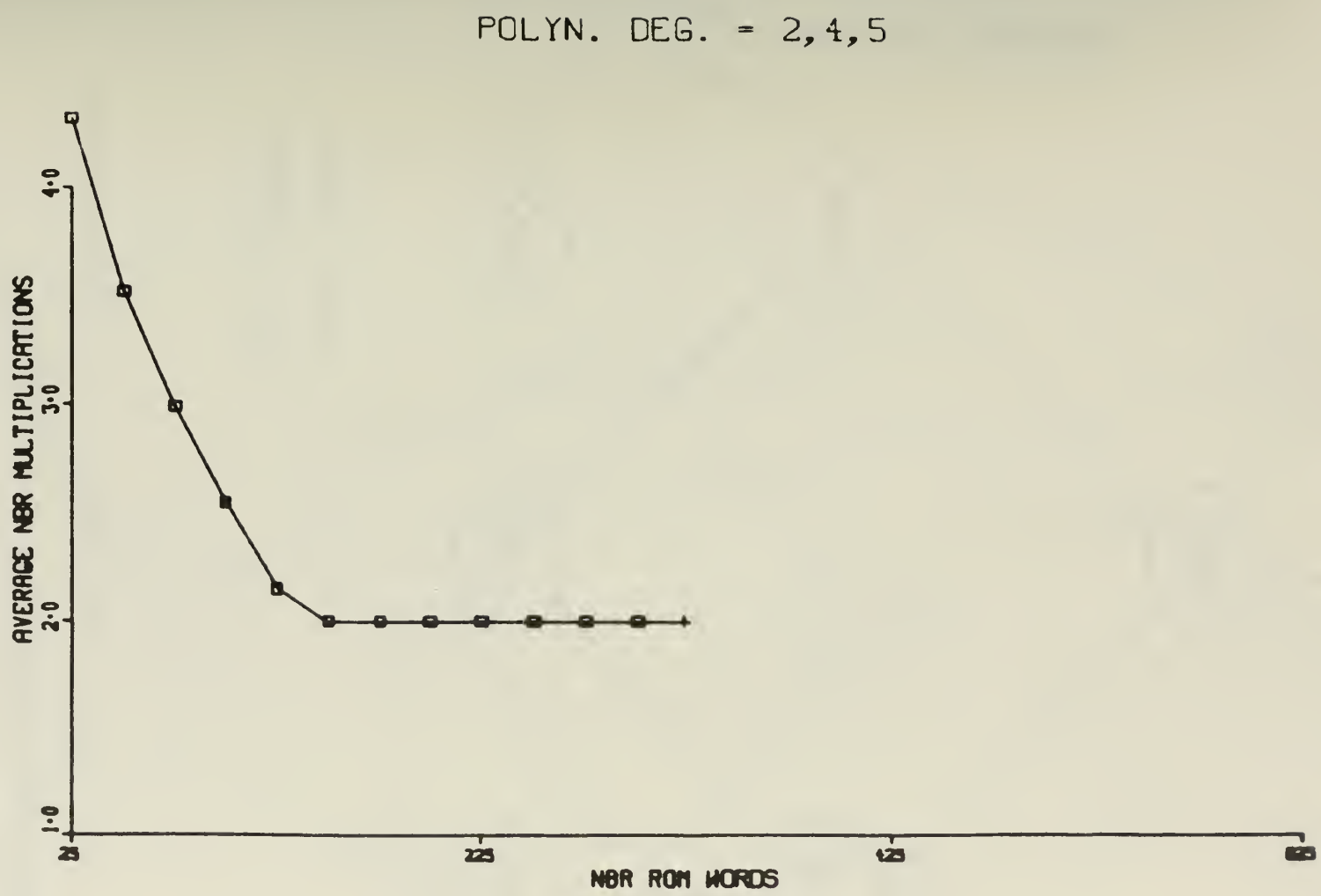


Figure 5.22 Optimal Curve: Average Number of Multiplications vs Cost in Number of ROM Words; VIVD Case; Two Joints; Polyn. Deg. = 2,4,5

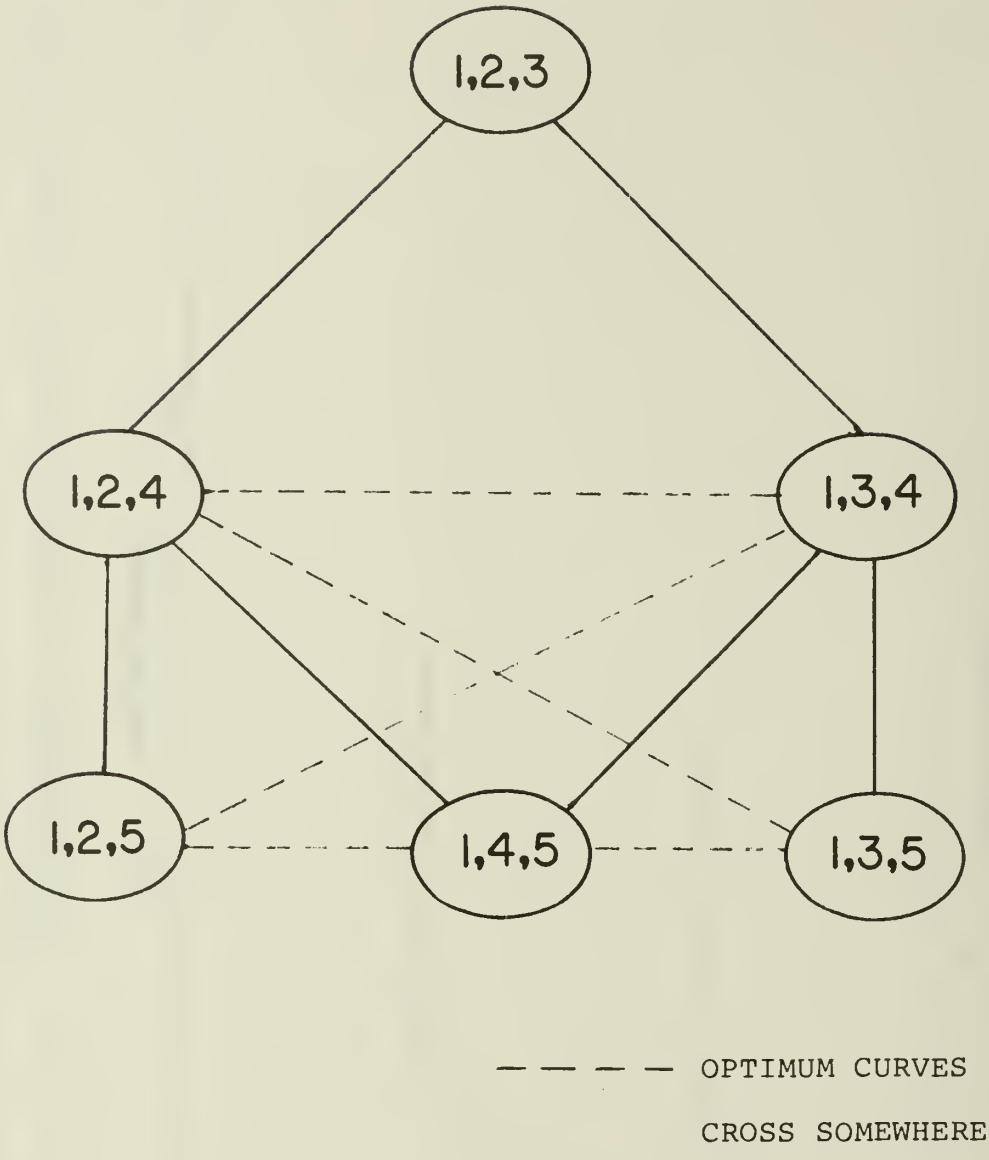


Figure 5.23 Ordering Relationship for Implementations Involving Three Polynomials (Two Joints)

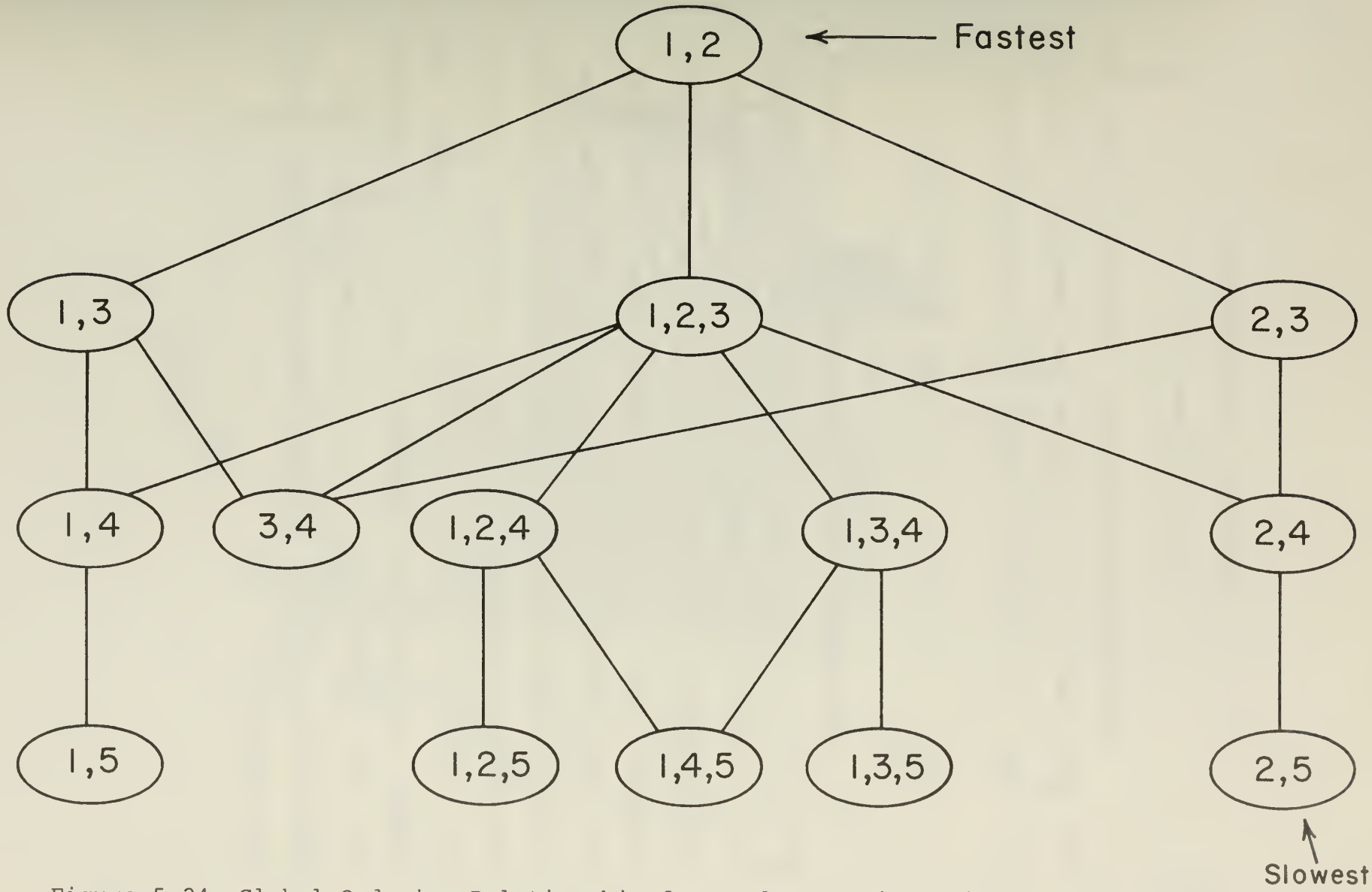


Figure 5.24 Global Ordering Relationship for Implementations with 2 or 3 Polynomials

5.2 The Effect of a Finite-Length Look-Up Address

The program was run using the same algorithm but with each iterative evaluation of the breakpoint interval $h_i = x_{i+1} - x_i$ replaced by the closest value satisfying:

$$2^{-k} \leq h_i \qquad [k \text{ is some integer here}]$$

This replaces each interval length by its closest power of 2 approximation. Intuitively what should happen is an increase of the number of ROM words for the same average number of multiplications. In other words, for a given cost, that is, a given maximum number of ROM words available, the speed should be smaller. This effect can be seen in Fig. 5.25 which can be compared with Fig. 4.12. The minimum is situated at (XJ(2) ≃ .7 and XJ(3) ≃ .8) in Fig. 4.12 whereas, for the same cost (R_0 = 350 Words), the minimum in Fig. 5.25 is situated at (XJ(2) = .55 and XJ(3) = .75). The implication of this is that fewer one or two degree polynomials are used and consequently slower evaluation results.

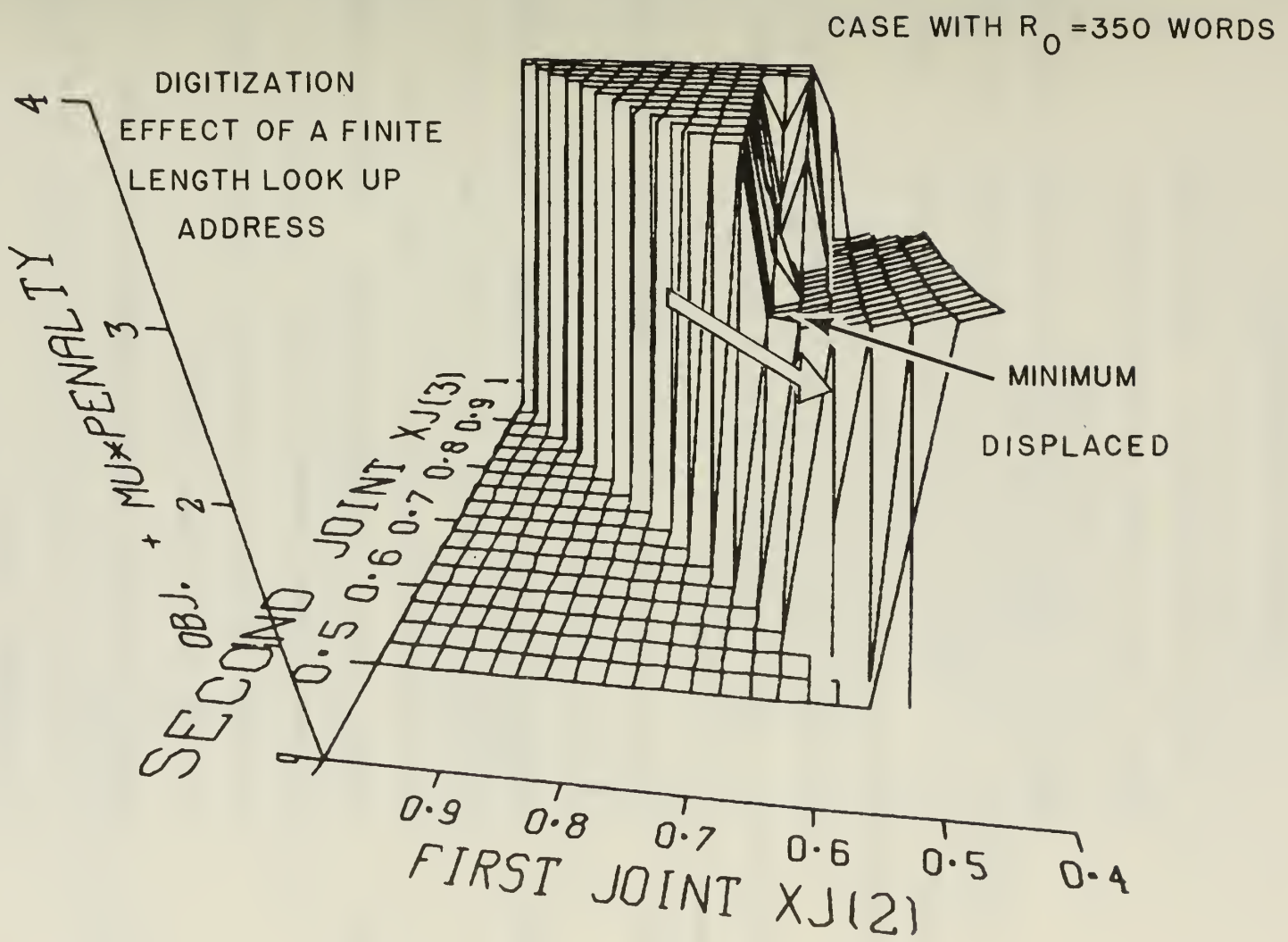


Figure 5.25 Effect of Digitization on the Position of the Minimum. Two Joints. Polynomial Degrees 1,2,3

6. HARDWARE IMPLEMENTATION

6.1 Design Considerations: Expected Time Delay. How to Store the Coefficients

Figure 6.1 shows a general structure for evaluating a function. The coefficients of the polynomials are stored in ROMs and the address decoder determines where to look up the coefficients from the argument x.

The algorithm may loop several times through the multiplier. We will distinguish four types of function generator architecture:

UIUD Uniform Interval, Uniform Degree: same degree for all the polynomials over the whole range.

VIUD Variable Interval, Uniform Degree: intervals will have different lengths to save ROM; same degree for all the polynomials over the whole range.

UIVD Uniform Interval, Variable Degree: intervals have the same lengths between joints, but different polynomial degrees are used over the whole range (but the same between joints).

VIVD Variable Interval, Variable Degree: the most general case; a combination of variable intervals and variable degrees.

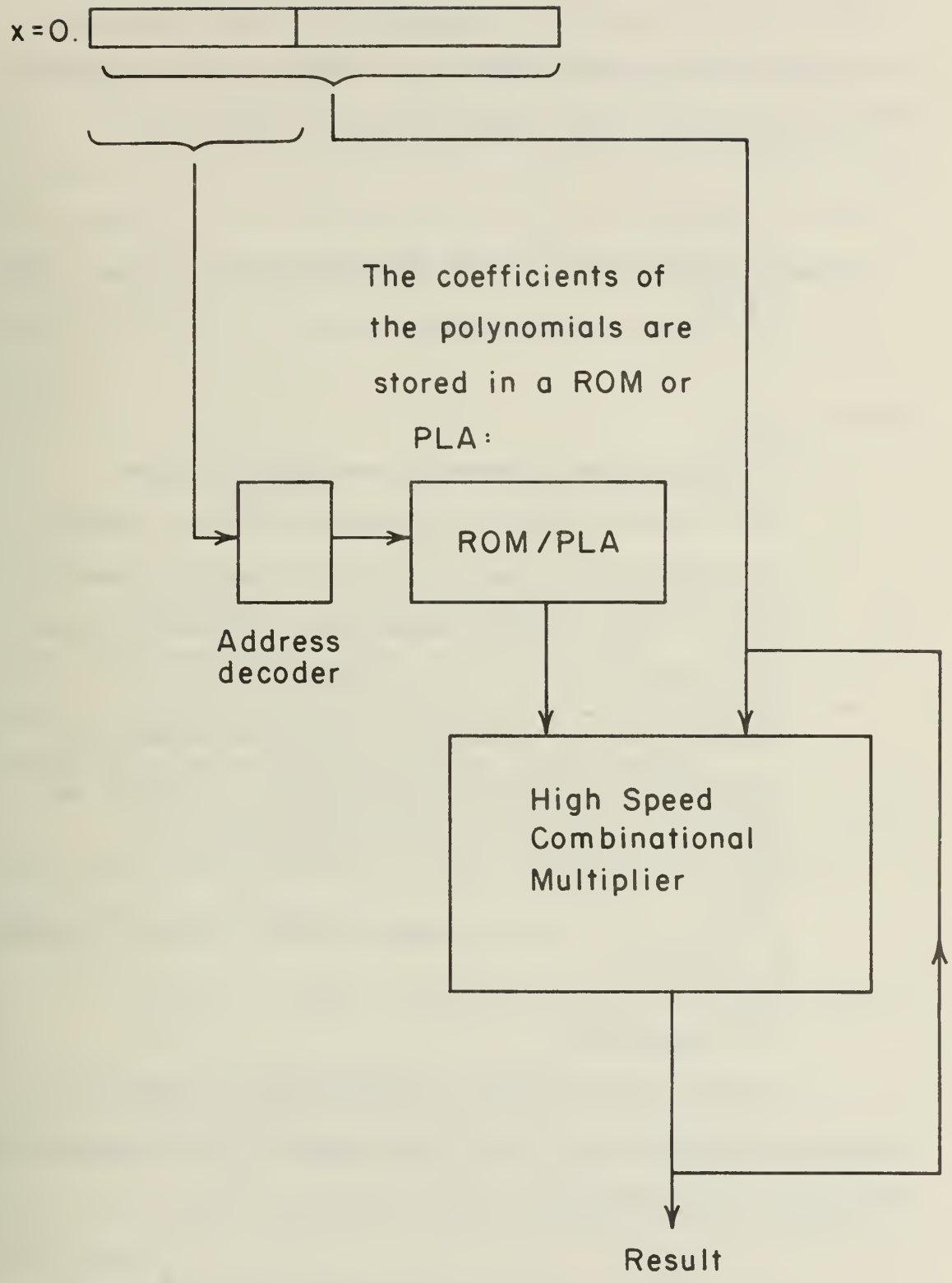


Figure 6.1 Sketch of the Arithmetic Unit for Function Evaluation

Before looking at the detailed design, we will evaluate the expected time delay. For parallel evaluation of the polynomial, the average delay is given by:

$$\tau_{avp} = \sum_{j=1}^{s} (2\log(m_j)\ T(n) + (1 \text{ ROM access time})_j) \int_{X_j}^{X_{j+1}} \frac{1}{x \ln \beta}\, dx$$

where

- s is the total number of joint partitions
- T(n) is the time to multiply two n-bit numbers
- the ROM access time must be counted only once in the interval j since subsequent accesses can be overlapped with the multiplication cycle.
- $2\log(m_j)$ is the number of levels necessary to evaluate a polynomial of degree m_j entirely in parallel, the number of multiplications.
- $\int_{X_j}^{X_{j+1}} \frac{1}{x \ln \beta}\, dx$ is the probability of finding argument x in interval j

If the evaluation of the polynomial of degree m_j is not performed in parallel the expression above should be written (serial case):

$$\tau_{avs} = \sum_{j=1}^{s} (m_j\ T(n) + (1 \text{ ROM access time})_j) \int_{X_j}^{X_{j+1}} \frac{1}{x \ln \beta}\, dx$$

If there is no address decoding and if we assume an implementation where the breakpoints are separated by $h_i = 2^{-\ell_j}$ or, in other words, where there is a direct look-up over $(\ell_j - 1)$ bits, the ROM access time is proportional to $\log_2(\ell_j - 1)$. A factor K_T will be introduced to take into account the technology used. If we assume that the multiplier uses the same technology as the ROM, we can write:

$$\tau_{avs} = \sum_{j=1}^{s} (m_j \; T(n) \cdot K_T + \log_2(\ell_j - 1) \cdot K_T) \ln \left[\frac{X_{j+1}}{X_j} \right]$$

Depending on the structure of the multiplier, T(n) can be proportional either to n or to logn. This formula, when K_T is known, can be used to find the average speed expected from a given technology and design.

The first ROM access is the most important because the first coefficient, a_m, must be obtained in order to begin the evaluation of

$$P_{m_j}(x) = (((a_m x + a_{m-1})x + a_{m-2})x + a_{m-3})x + a_{m-4} \cdots$$

While $a_m x$ is being computed, a_{m-1} can be retrieved and added to $a_m x$. While $(a_m x + a_{m-1})x$ is being computed, a_{m-2} is looked up and the time sequence can be viewed as follows:

Operation		Operation done in parallel
ROM access	look up a_m	--
Multiply	$a_m \cdot x$	look up a_{m-1}
Add	$a_m x + a_{m-1}$	look up a_{m-2}
Multiply	$(a_m x + a_{m-1})x$	
Add	$(a_m x + a_{m-1})x + a_{m-2}$	look up a_{m-3}
Multiply	$((a_m x + a_{m-1})x + a_{m-2})x$	
Add	$((a_m x + a_{m-1})x + a_{m-1})x + a_{m-3}$	look up a_{m-4}
	$\vdots$	$\vdots$

It could be advantageous therefore to use high speed ROM for storing the first coefficients (the a_m's) and slower ROM for all the other coefficients. The maximum access time for the slower ROM can be roughly taken to be the time to multiply two n-bit numbers plus one add time. Thus the bulk of ROM look-up memory can be realized with cheaper and slower ROM.

How to Store the Coefficients

When coefficients a_m, a_{m-1}, a_{m-2} . . . have been computed, the straightforward approach is to store them directly, and to evaluate $P_m(x)$ using the generalized Horner's rule. Assuming a second degree polynomial for example:

$$P_2(x) = a_2x^2 + a_1x + a_o$$

One could store a_2, a_1 and a_o as n-bit words and compute

$$P_2(x) = (a_2x + a_1)x + a_o$$

However, this is not the best way to store the coefficients for evaluating P_m. Substantial memory can be saved by noticing that:

$$x = x_i + 2^{-\ell}\,\delta \quad \text{with } 0 \leq \delta < 1 \quad \text{(see Fig. 3.1)}$$

$P_2(x)$ can now be written

$$P_2(x_i+2^{-\ell}\delta) = A_2\ 2^{-2\ell}\delta^2 + A_1\ 2^{-\ell}\delta + A_0$$

$$\text{where } A_2 = P_2(x_i),\ A_1 = P_2'(x_i),\ A_0 = \frac{P_2''(x_i)}{2!}$$

Now:

- A_0 must have n bits precision
- A_1 must have only $(n-\ell)$ bits precision since greater precision would be lost. This is because A_1 in the term $A_1\ 2^{-\ell}\delta$ is multiplied by the factor $2^{-\ell}$ which causes it to be significant only in the ℓth bit position. Thus only precision $(n-\ell)$ is required. This statement must be slightly modified however when the magnitude of A_1, which depends on $f(x)$, is considered.
- A_2 must have (n-2) bits for the same reason.

Let us see how much memory is actually saved when the degree m of P_m is large:

Assuming that $P_m(x_i + \delta 2^{-\ell}) = \sum_{k=0}^{m-1} A_k\ \delta^k$ and that ℓ bits of lookup are used with

$$A_k = \frac{P^{(m-k)}(x_i)}{k!}$$

A_0 must be n bits wide

A_1 must be $n-\ell$ bits wide

.
.
.

A_{m-1} must be $n-(m-1)\ell$ bits wide

The total number of words used is thus:

$$N = n + (n-\ell) + (n-2\ell) + \ldots + (n-(m-1)\ell)$$

$$N = m\,n - m\,\frac{(m-1)}{2}\,\ell = m\,n\,(1-\frac{(m-1)}{2n}\,\ell)$$

This diminishes the amount of ROM needed by a relative value of $\frac{(m-1)}{2n}\,\ell$. If $m \simeq \frac{n}{\ell}$, the savings in Memory can be up to 50% compared to a straightforward storage scheme.

In addition, only an n by $(n-\ell)$ multiplication need be performed rather than an (nxn) multiplication.

6.2 Architectural Considerations: Obtaining the Address from the Argument

6.2.1 UIUD Case

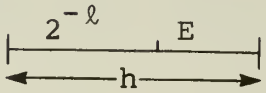
In this case both the breakpoint interval length and the degree of polynomial approximant are uniform over the whole range. The value of h satisfying

$$C(f,P_m,x_i,x_i+h) \leq \varepsilon_0 \quad x_i \varepsilon \left[\frac{1}{\beta}\,,\ 1\right]$$

must be determined first. This h must be the most "stringent" h over [a,b] so that the value of the criterion

C will always be less than ε_o. h can be written:

$$h = 2^{-\ell} + E \qquad \text{(6.1)}$$



where

$$E < 2^{-\ell}$$

Stated another way, the number of ROM address lines necessary to look up the coefficients of the polynomial approximant will be:

$$\ell = \lceil \log \frac{1}{h} \rceil$$

where x is the "ceiling function," the smallest integer greater than or equal to x. In the case of $\beta = 2$ and a normalized floating point number, the most significant digit is 1 and the number of ROM address input bits is then $(\ell - 1)$. Figure 6.2 shows the essential elements of the UIUD hardware realization.

When the E defined in (6.1) is very close to $2^{-\ell}$, a large amount of memory is wasted using a UIUD design and a VIUD design, as described in the following section, may be more economical.

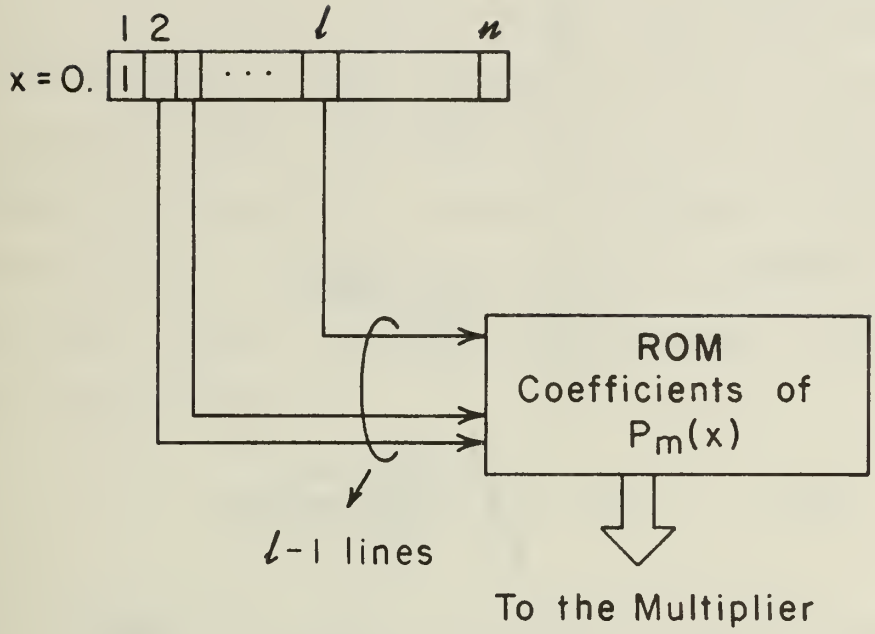


Figure 6.2 Type 1 Architecture: UIUD

6.2.2 VIUD Case

In this case each interval $[x_i, x_{i+1}]$ has a length h_i dependent on the value of the function g(x) defined in Section 3.3.2:

$$h_i = x_{i+1} - x_i = k_m \, g(x_i)$$

where

$$k_m = \left[\frac{\varepsilon_o^2}{K_m}\right]^{\frac{1}{2m+3}}$$

and

$$K_m = \int_0^1 \frac{\prod_{k=0}^{m} (r-\alpha_K)^2}{((m+1)!)^2} \, dr$$

$$g(x) = \{(f^{(m+1)}(x_i))^2 w(x_i)\}^{\frac{1}{2m+3}}$$

h_i depends essentially on the behavior of the (m+1)st derivative of the function f.

The rate of change of h_i as a function of x is $\frac{dh}{dx}$. Assuming for simplicity $w(x) \equiv 1$

$$\frac{dh}{dx} = -\frac{2k_m}{2m+3} \cdot \frac{f(x)^{(m+2)}}{[f(x)^{(m+1)}]^{\frac{2m+5}{2m+3}}}$$

Example:

$$f(x) = \frac{1}{x} \qquad m = 1$$

$$\frac{dh}{dx} = \frac{8k_m}{5} \frac{1}{x} \left(\frac{2}{x^3}\right)^{-\frac{2}{5}}$$

When the number of breakpoints is large h increases approximately as the continuous function $\frac{dh}{dx}$. The region where the size of h changes the most rapidly is at the left side of the interval. As in the UIUD case, it is not possible to implement the look-up for a general (infinite precision) h_i since locating the two breakpoints close together would require many address lines. The number of ROM address lines is, for the smallest h:

$$\ell - 1 = [\log \frac{1}{h}] - 1$$

However, the smallest value of h need not be used over the entire interval. The smallest h is at x = .5 in the case of 1/x. (See Fig. 6.3) When x_i increases from .5 to 1 in this example, the length h_i increases. At point P_2 the value of h satisfying $C(f,P_n,x_i,x_i+h_i) = \varepsilon_o$ becomes twice the smallest value $2^{-\ell}$. At points P_3, $h = 3x2^{-\ell}$ and at point P_j, $h = j\ 2^{-\ell}$. It is much easier to detect integral multiples of $2^{-\ell}$ than fractional increments in $2^{-\ell}$.

The total number of ROM addresses is

$$A(.5,1) = \sum_j \frac{P_{j+1} - P_j}{j2^{-\ell}} \qquad P_j \leq 1$$

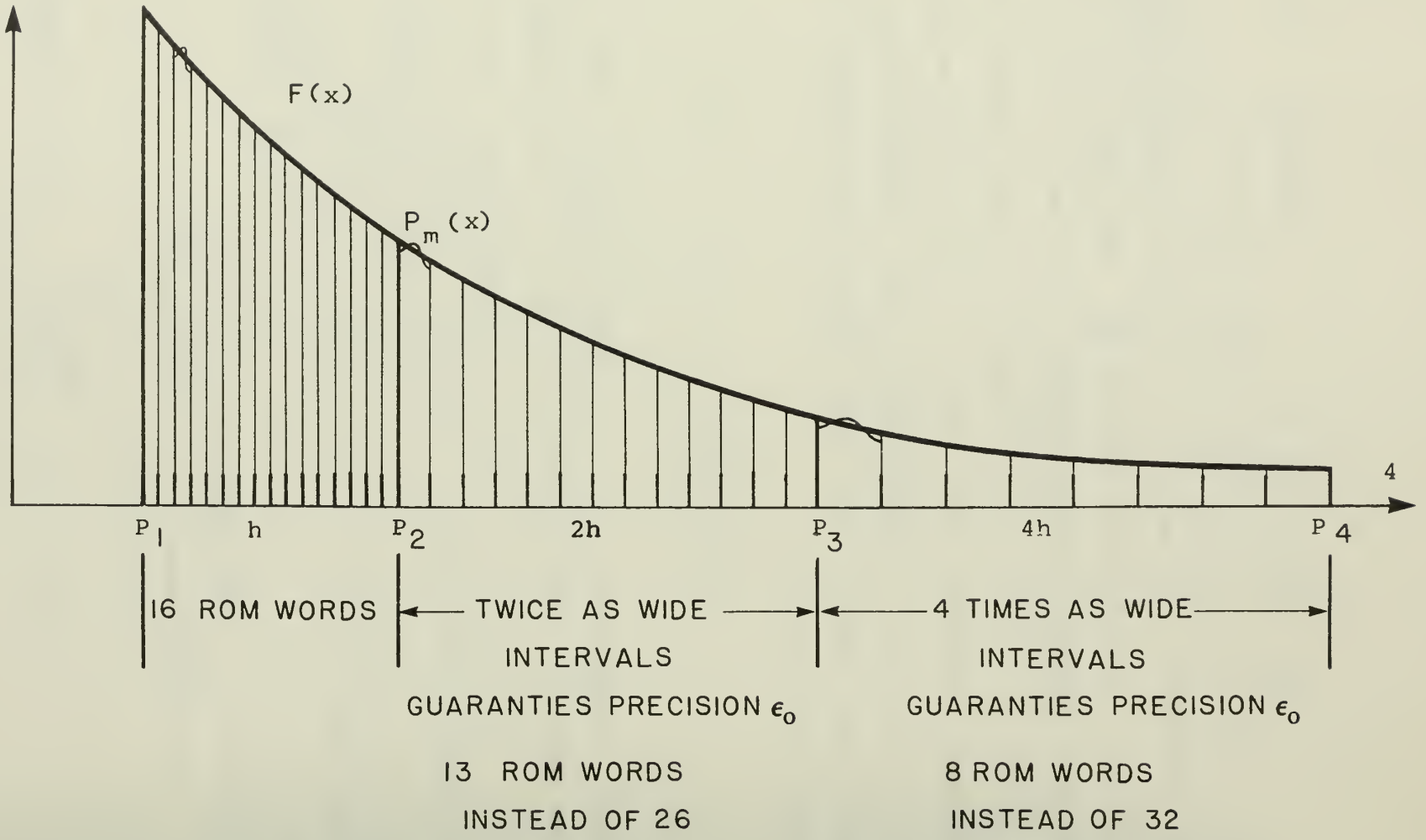


Figure 6.3 VIUD Case; Variation of the h's Over the Interval

The number of address lines is $[\log_2 A(P_1,1)]$. The abscissas P_j are such that:

$$k_m\, g(P_j) = j\, 2^{-\ell}$$

If we want to determine how many such points P_j there are between .5 and 1, simply compute the ratio:

$$\frac{k_m g(1)}{k_m g(.5)}$$

and if $k_m g(.5) = 2^{-\ell}$ (this depends on ε_o) then the ratio:

$$\log \frac{g(1)}{g(.5)}$$

will give an approximate value of the numbers of such points P_j.

Example:

$$f(x) = \frac{1}{x} \qquad m = 1 \qquad w(x) = 1 \qquad \beta = 2$$

$$g(x) = \left(\frac{2}{x^3}\right)^{-\frac{2}{5}}$$

$$g(1) = 2^{-.4} = 1.3195$$

$$g(.5) = 2^{-8/5} = .3928$$

$$\frac{g(1)}{g(.5)} \simeq 4$$

$$\log\left(\frac{g(1)}{g(.5)}\right) \simeq 2$$

So we have 2 such points for the above example.

The standard way for decoding the addresses would then be to introduce a ROM decoder between the ℓ-1 most significant bits of the argument x and the address lines (ℓ') of the coefficient ROM. Each value of h must satisfy the condition

$$h = j.2^{-\ell}$$

for feasibility of implmentation in hardware (see Fig. 6.4).

If one is willing to consider only points P_j such that

$$h = 2^J \cdot 2^{-\ell},$$

then the ROM decoding used above can be advantageously replaced by a barrel shifter (Fig. 6.5). When the interval detected is such that $h = 2.2^{-\ell}$, it is necessary to increment by one, using only every other word. This can be done by looking up bits 1, 2 . . . (ℓ - 1), discarding bit ℓ. When $h = 4.2^{-\ell}$, bit (ℓ - 1) is discarded and bits 1, 2, . . . (ℓ - 2) are used for direct look-up. The barrel shifter allows a combinational shifting of a string of input bits (Fig. 6.5). The number of shifted places is controlled by independent inputs and the propagation bit can be chosen to be 0 or 1. The control for the shift amount is decoded

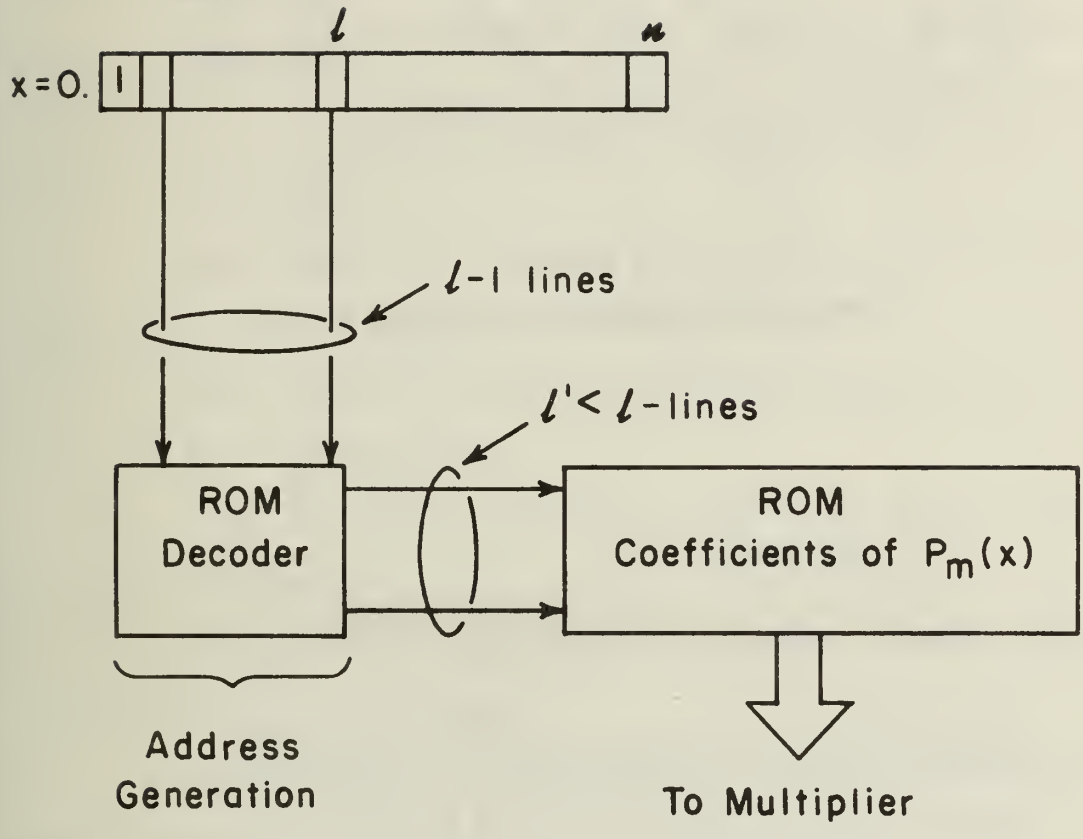
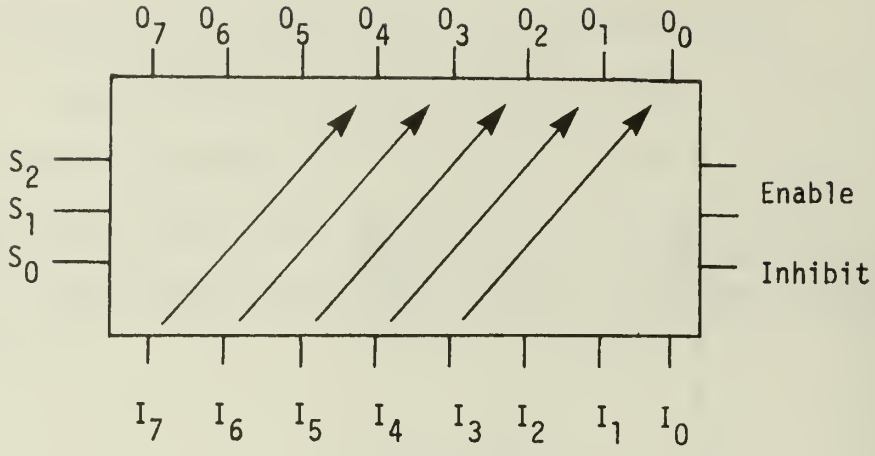


Figure 6.4 Type 2 Architecture: VIUD



Example with a shift of 3 ($S_0S_1S_2$ = 110)

Truth table :

INHIBIT	ENABLE 1 & 2	S_0	S_1	S_2	O_0	O_1	O_2	O_3	O_4	O_5	O_6	O_7
0	1	0	0	0	$\bar{I}_0$	$\bar{I}_1$	$\bar{I}_2$	$\bar{I}_3$	$\bar{I}_4$	$\bar{I}_5$	$\bar{I}_6$	$\bar{I}_7$
0	1	1	0	0	$\bar{I}_1$	$\bar{I}_2$	$\bar{I}_3$	$\bar{I}_4$	$\bar{I}_5$	$\bar{I}_6$	$\bar{I}_7$	1
0	1	0	1	0	$\bar{I}_2$	$\bar{I}_3$	$\bar{I}_4$	$\bar{I}_5$	$\bar{I}_6$	$\bar{I}_7$	1	1
0	1	1	1	0	$\bar{I}_3$	$\bar{I}_4$	$\bar{I}_5$	$\bar{I}_6$	$\bar{I}_7$	1	1	1
0	1	0	0	1	$\bar{I}_4$	$\bar{I}_5$	$\bar{I}_6$	$\bar{I}_7$	1	1	1	1
0	1	1	0	1	$\bar{I}_5$	$\bar{I}_6$	$\bar{I}_7$	1	1	1	1	1
0	1	0	1	1	$\bar{I}_6$	$\bar{I}_7$	1	1	1	1	1	1
0	1	1	1	1	$\bar{I}_7$	1	1	1	1	1	1	1
1	x	x	x	x	1	1	1	1	1	1	1	1
x	0	x	x	x	1	1	1	1	1	1	1	1

Figure 6.5 Barrel Shifter

(with low precision) directly from the input x (Fig. 6.6). The shifted number is now the address of the ROM. The ROM savings to be expected this way will be less than for the general case, but are balanced by the reduction in control and the simplicity of design.

For large N, however, this technique is not advantageous for all types of functions (see Appendix A.3).

6.2.3 UIVD Case

When there are several polynomials and the intervals are chosen to be uniform the designer has

a. the flexibility of a variable average number of multiplications
b. the advantage of uniform intervals, which is easier to decode.

Figure 6.7 illustrates the general organization of a UIVD function generator.

By choosing the largest h satisfying the criterion and having $h = 2^{-J}$, direct look-up using the argument is feasible. A decoder of low precision will determine the number of multiplications to be performed and which bank of ROMs to read from. This decoder can be implemented with a small ROM. By choosing the same joints for several functions, one can choose the same addressing scheme (See Section 8).

This structure is the most appealing for its flexibility and ease of implementation.

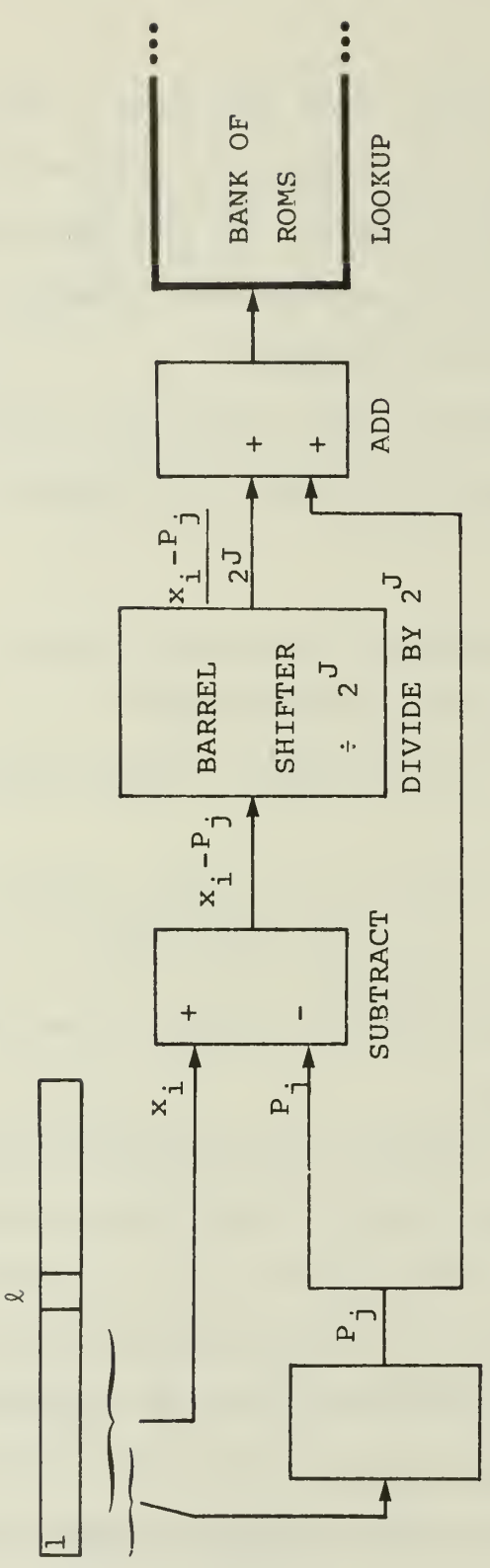


Figure 6.6 Address Decoding Using a Barrel Shifter

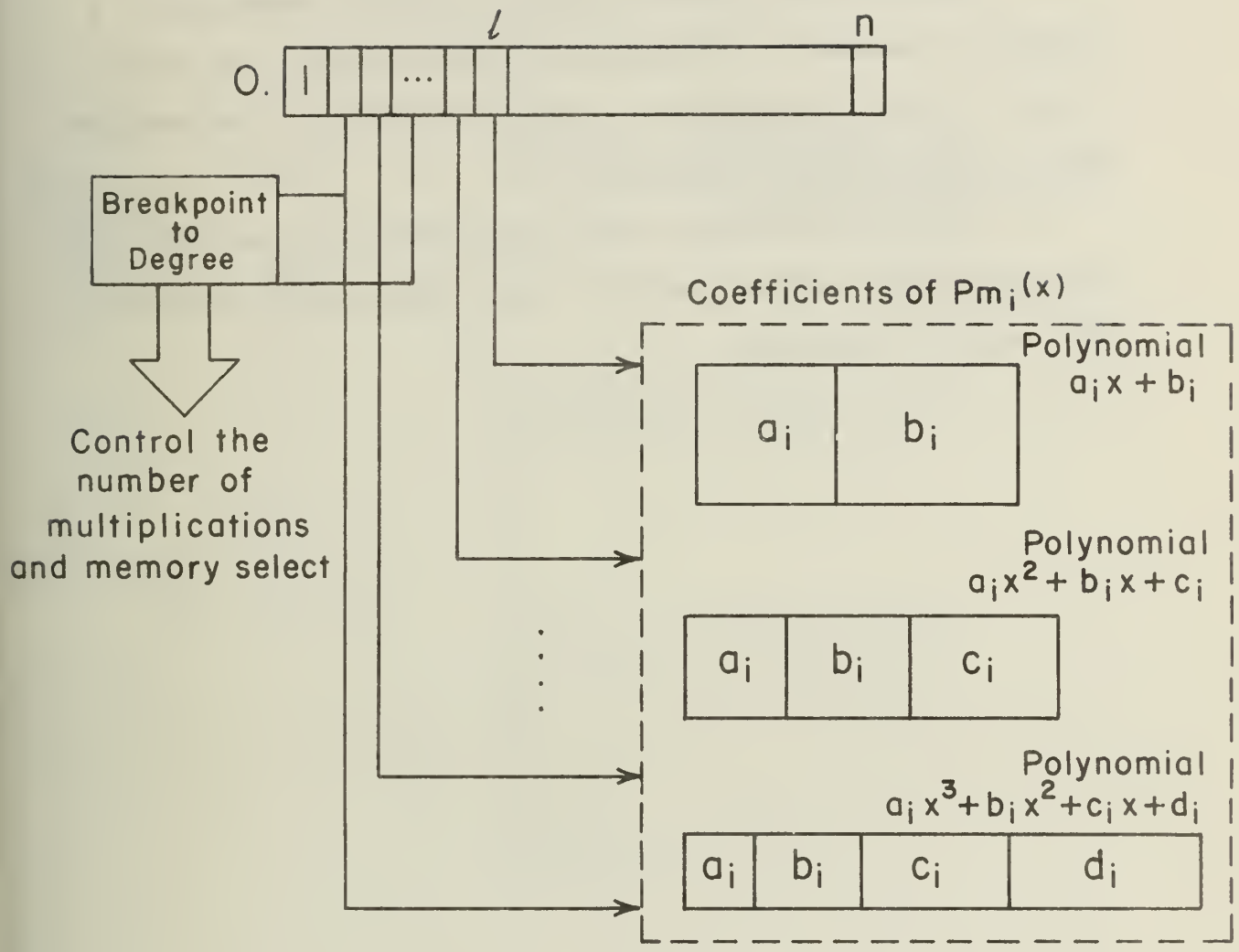


Figure 6.7 Type 3 Architecture: UIVD

6.2.4 VIVD Case

This case is illustrated in Fig. 6.8.

Due to the variable intervals within each joint, a decoder is needed for each bank of ROMs storing the coefficients of the polynomials of different degree. The decoder can be implemented with ROMs.

This architecture would be interesting for very high precision implementations where substantial savings in ROM could be achieved.

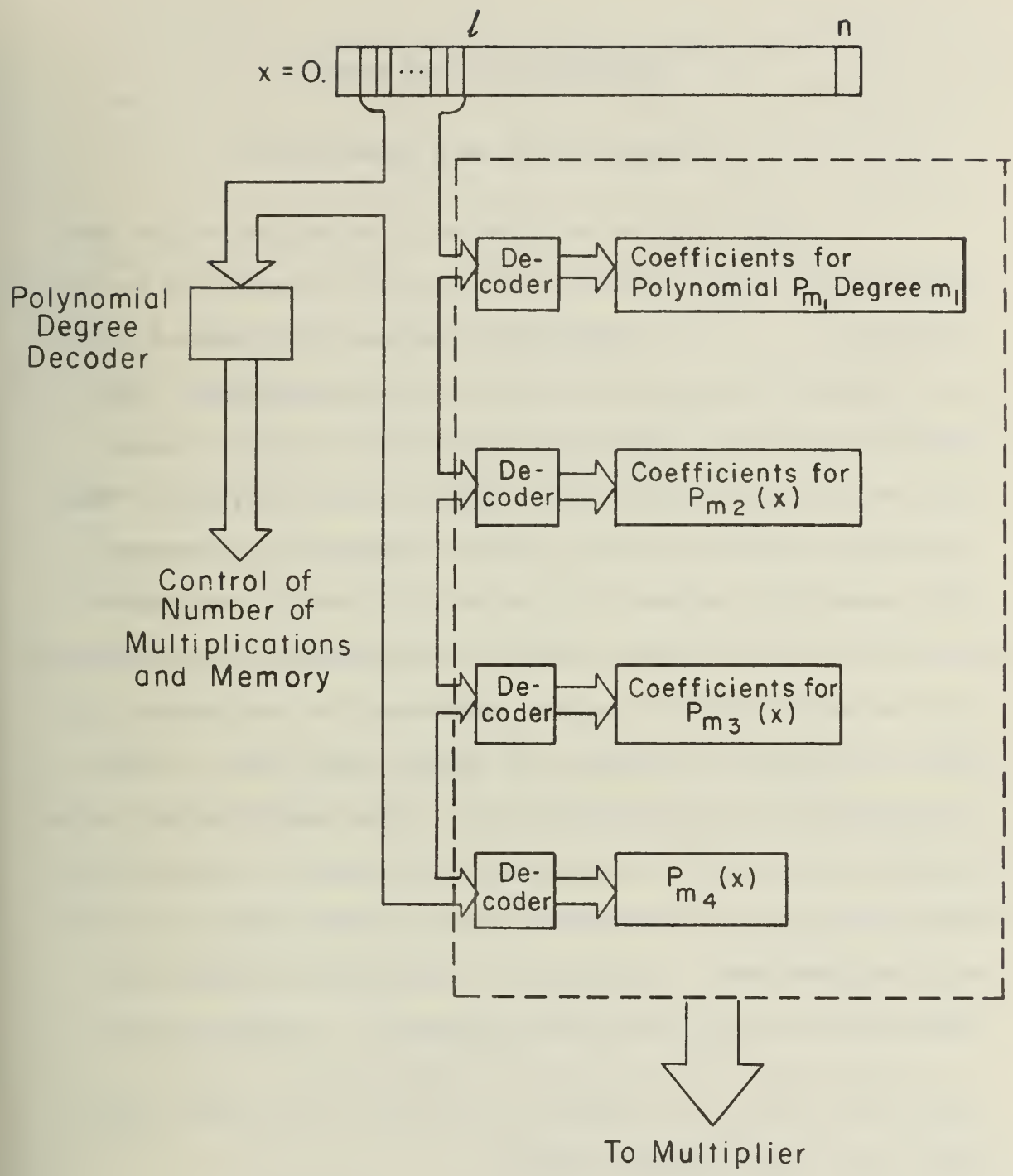


Figure 6.8 Type 4 Architecture: VIVD

7. COMPLEXITY OF THE METHOD

7.1 Introduction and Background

In this section we consider the bounds on the ROM-based method (the R-method) for function generation as the precision $n \to \infty$. We ultimately compare the R-method with the classical single polynomial evaluation scheme. This will be referred to as the "S-method."* We will assume that an unlimited number of processors is available. This requires a brief review of what is meant by "processor" when memory is allowed to grow with n. We are concerned with order of magnitude results and multiplicative constants are ignored. Multiplicative constants for the bounds are left for further research. We assume that f(x) is to be evaluated with error $0(2^{-n})$ as $n \to \infty$ for any floating point number x with an n-bit fraction in a finite interval [a,b]. The function f(x) is continuously differentiable in [a,b] and "well-behaved" in general. In the derivations below the notation $f \approx g$ means that $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 1$ and $f = 0(g)$ means that there is a positive constant C such that $f(n) < Cg(n)$ for n sufficiently large.

---

*We will also present a bound for the time to evaluate f(x) which is asymptotically equivalent to a bound given by Brent BRE76 and is obtained by using a number of ROM words that does not grow exponentially but rather in polynomial space.

Before starting the study of the complexity of the R-method as $n \to \infty$ we will briefly review the measure of effectiveness of an algorithm and the best known bounds for parallel polynomial evaluation and multiple precision function evaluation.

7.1.1 The Measure of Effectiveness of an Algorithm

Kuck [KUC74] defines speed-up of a parallel algorithm A over a serialone a as $S_A = \frac{Ts}{T_A}$ where T_A is the number of time units required to perform some calculation using p independent processors and T_s is the time required to do the same calculation on a single processor of the same type. The definition of processor depends on the algorithm considered. It can be a CPU, a hardware multiplier, an adder or even a single gate. The efficiency is defined as $\varepsilon = S_A/p_A(\leq 1)$. Because speed-up alone is not an adequate measure for comparing algorithms, Kuck defines the cost of an algorithm using p processors as $C_A = p_A T_A$. The measure of the effectiveness of an algorithm is then given by the ratio: $E = S_A/C_A$. A high cost counterbalances a high speed-up factor.

To compare two algorithms A and B that perform the same function, it is necessary to compare $\frac{S_A}{C_A}$ and $\frac{S_B}{C_B}$. A is better than B if $\frac{S_A}{C_A} \Big/ \frac{S_B}{C_B}$ is greater than 1. This ratio can be seen to be equal to $\frac{p_B}{p_A} \frac{T_B{}^2}{T_A{}^2}$. As a result, a parallel

algorithm can be characterized by 4 numbers: speed-up, efficiency, cost and effectiveness. As $n \to \infty$, the limits of these 4 numbers can be studied and constitute an adequate and compact means of comparing algorithms.

7.1.2 The Best Known Bounds for Parallel Polynomial Evaluation

Several algorithms for the parallel evaluation of arithmetic expressions have been proposed. The special case of polynomial evaluation was investigated by Maruyama [MAR 73] and Munro and Paterson [MUN73]. They essentially showed that an upper bound for the number of steps required to evaluate polynomials of degree μ is given by:

$$(1+\varepsilon)\log\mu$$

where

$$\varepsilon \sim \sqrt{\frac{2}{\log\mu}}$$

In other words, a μth degree polynomial can be evaluated in $\log\mu + 0(\log\mu)^{\frac{1}{2}}$ steps. This bound assumes that multiplication and addition times are equal and that arbitrarily many processors are available. The number of processors required to attain this bound is 2μ.

When the number of processors is bounded and finite, Snir and Barak [SNI77] have shown that any polynomial can be computed in $\frac{3\mu}{2p+1} + 0(p^2)$ steps with p processors.

7.1.3 Notation and Assumptions

The R-method (ROM-method) as we define it for this study is illustrated in Figs. 7.1 and 7.2. f(x) is to be evaluated with precision $0(2^{-n})$ as $n \to \infty$ over a suitable interval [a,b]. We are interested in the case where n is much greater than the word-length of the machine so that the mantissa occupies several single precision words. Thus, when we say that a "word" of Read-Only Memory has size 0(n), we mean that it would take 0(n) single precision words to store such a "word." We assume a UIUD approach. The VIVD improvement affects the bounds by multiplicative constants as is shown in Appendix A-3. The polynomial approximants have degree μ. Because the degrees can be made dependent on n as $n \to \infty$, we will accordingly denote by $\mu(n)$ the degrees of the polynomial approximants. The total number of Read Only Memory words necessary to store all the polynomial coefficients will be denoted by $\mathcal{M}(n)$.

The number of distinct memory addresses is R(n) and differs from $\mathcal{M}(n)$ by the product of the number of required coefficients times the number of words, $n\mu(n)$. We assume binary representation. Each polynomial $P_{\mu(n)}$ can be evaluated in two ways. One can assume a number

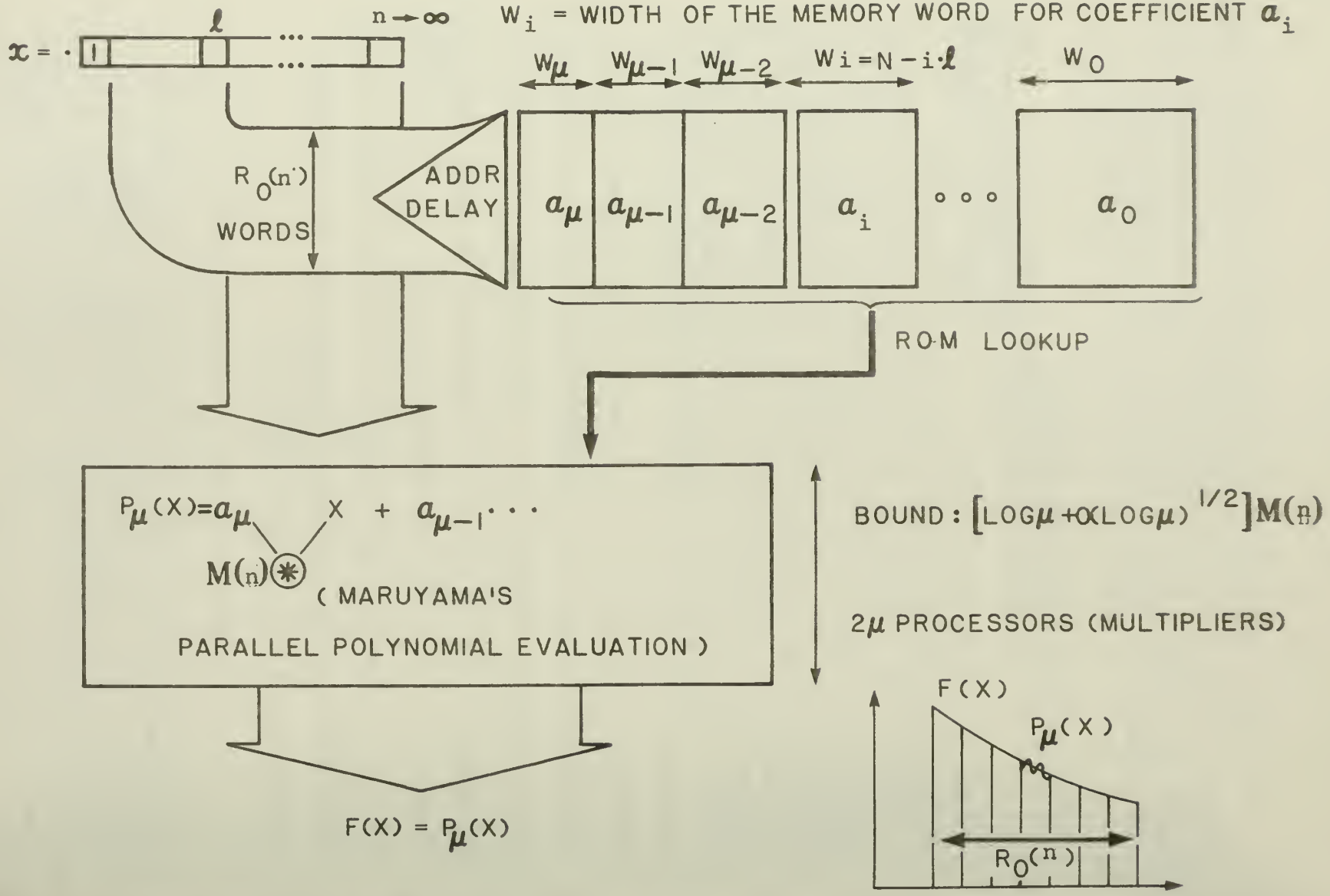


Figure 7.1 The R-Method

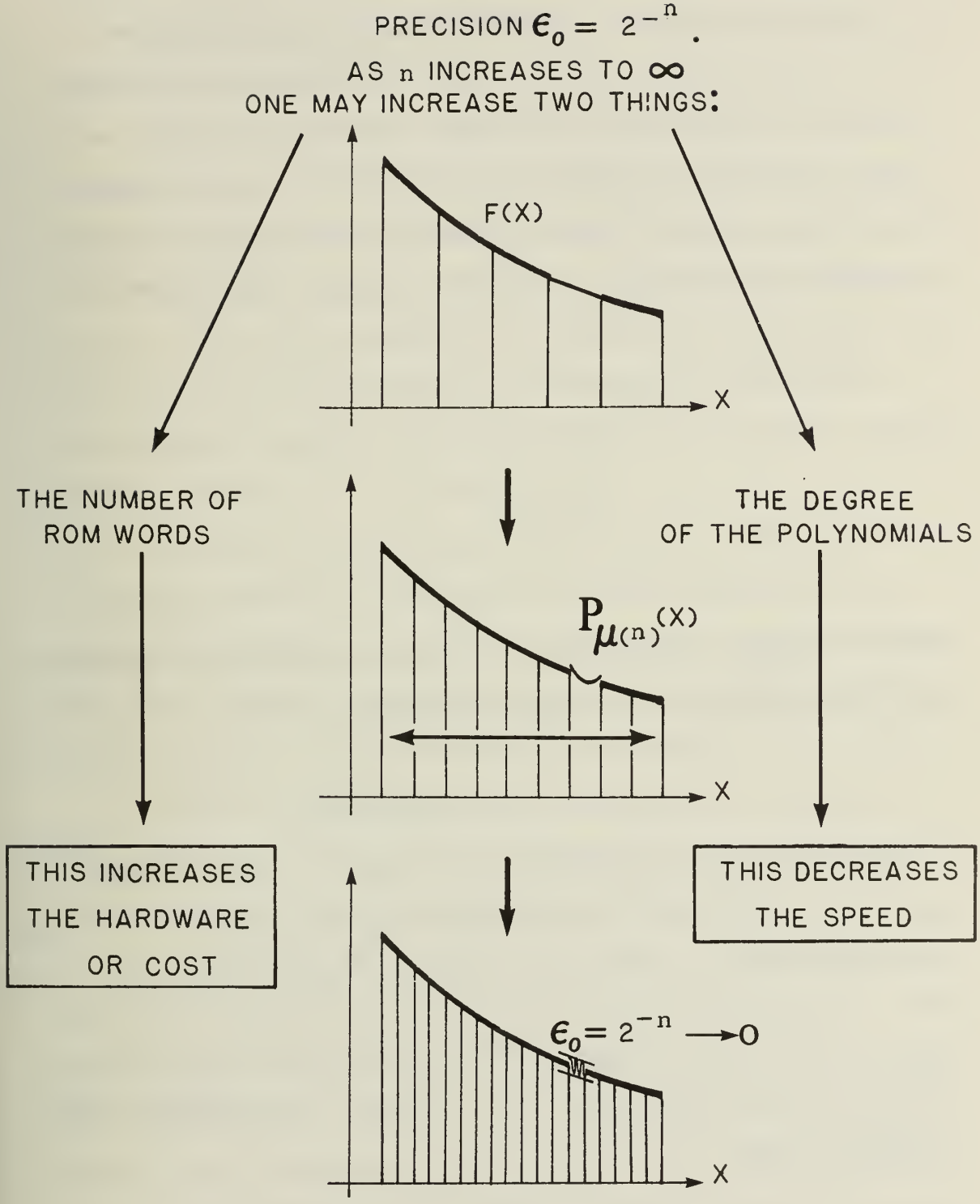


Figure 7.2 Tradeoffs when $n \to \infty$

of processors limited to p for evaluating $P_{\mu(n)}$ or else an unlimited number of processors may be available as $n \to \infty$. A processor takes a step time $\tau(n)$. In our case $\tau(n)$ will be the time to multiply two n-bit integers using single precision multiplications is $0(n^2)$. It has been shown that a better time is possible by neglecting terms which do not contribut to precision 0(n). One bound is

$$(n)=O(n^{1.58..})$$

The best known bound however has been obtained by Schönhage and Strassen [SCH71]. They showed that

$$(n)=O(n.\log(n).\log\log(n)) \tag{7.1}$$

is the time to multiply two n-bit numbers using a single precision multiplier and the result has precision n. It has been conjectured by Knuth that this is the optimum bound.

Thus, we assume in what follows that the time to add/multiply two numbers is $\tau(n)$ given by the Schönage and Strassen bound.

7.1.4 Time vs space. An Intuitive Look at the R-Method

We would like to pause here before going on to a formal analysis, and present an informal discussion about a question which, in a sense, is a key point in the area of function evaluation in a computer. One must ask whether or not it is at all worthwhile to use Read Only Memory for computing f(x) rather than the traditional algorithmic, one-polynomial scheme. In other words, is it worth trading time for memory?

We would like to point out that this question is of general theoretical interest and that Hopcroft, Paul and Valiant [HOP77] in a paper entitled "On Time vs Space" have very recently presented an important result related to this problem. Their theorem states that:

"Every deterministic multitape Turing Machine of time complexity t(n) can be simulated by a deterministic Turing machine of tape complexity t(n)/logt(n)."

More research is needed in order to determine the exact relationship between this important theoretical result and the more specific problem of function generation using Read Only Memory. However, the reader will not fail to notice the striking similarity between the above theorem and the appearance of the bound 0(n/logn) scattered in our derivations on the complexity of the R-method.

The R-method has an easily implementable advantage that other methods do not have: the evaluation of f(x) can be speeded up two ways, one of which consists of using memory:

a) by evaluating $P_\mu(x)$ in parallel (many multipliers)

or b) by having more polynomials of lower degree, namely by doing more look-up (many memory words).

If n is fixed and the number of ROM words is fixed, we have seen that there is an optimum choice for the polynomials that yield the lowest average number of multiplications.

When f is approximated over [a,b] by a single polynomial, the only storage required is for the coefficients of this one polynomial. This simplicity is penalized by the necessity of using a high degree polynomial in order to maintain the required precision 2^{-n} over the entire interval. The speed of evaluation for this S method can be improved in only one way: by evaluating this single polynomial in parallel. What happens when $\varepsilon_o(n) = 0(2^{-n})$ and $n \to \infty$?

In the S-method, the only way to maintain a precision $\varepsilon_o(n)$ is to increase the degree of the polynomial approximant. The rate of increase of the degree will depend on the function f(x) and will have the effect of slowing down the evaluation of f(x) by increasing the number of multiplications.

In the R-method, one can modify both the number of ROM words and the degree of the piecewise polynomials to maintain the precision $\varepsilon_o(n) = 0(2^{-n})$. There is a relationship between the degree of the polynomials $\mu(n)$ and the total number of ROM words $m(n)$ necessary to guarantee $\| f - P_{\mu(n)} \| < 2^{-n}$. When the number of ROM words increases, this has two opposite effects:

a) it improves the speed-up of the method, and

b) it increases the cost of the computation.

What it does to the effectiveness (as defined by Kuck) is a question that we will try to answer in the next sections. Whether there is a strategy $[\mu(n), m(n)]$ that will make the effectiveness higher than the single-polynomial, parallel evaluation scheme for all n as $n \to \infty$ and still guarantee a precision $0(2^{-n})$ is also a fundamental question. If this were known, one could choose the best growth rate $0(R(n))$, matched with the appropriate degrees $0(\mu(n))$ to make the evaluation the most effective. It is possible that, as $n \to \infty$, it may not be feasible to keep the effectiveness of the R-method higher than the effectivenss of the S-method. In other words, the increase in speed obtained by adding memory may not "be worth" the increase in cost of memory.

In order to answer these questions we must express the following items as functions of n:

a) the relationship between $\mu(n)$ and $R(n)$

b) the speed-up of the R-method: S_R

c) the cost of the R-method: C_R

d) the effectiveness of the R-method: S_R/C_R

e) the ratio of effectiveness of R and S-methods $\frac{S_R}{C_R} \Big/ \frac{S_S}{C_S}$ and study it as $n \to \infty$.

7,2 Memory Complexity of the R-Method

We start from the relation

$$f(x)-P_{\mu}(x)= \prod_{i=0}^{\mu(n)} \frac{(x-x_i)}{(\mu+1)!} \quad f^{(\mu+1)}(\xi)$$

If the x_i's are chosen to be the roots of the Chebyshev polynomials of degree $\mu + 1$ then $\| f - P_{\mu} \|$ can be bounded:

$$|f(x)-P_{\mu}(x)| < \frac{h^{\mu+1} M_{\mu+1}}{2^{\mu} (\mu+1)!} \qquad (7.1)$$

where $M_{\mu+1}$ is a bound on the $(\mu + 1)$st derivative of $f(x)$.

At this point a closer look at the behavior of the $(\mu + 1)$st derivative of $f(x)$ as $\mu \to \infty$ is necessary. Before proceeding, we introduce some definitions.

7.2.1 Definition of Type 1 and Type 2 Functions

Definition 1

A continuously differentiable function f(x) over an interval [a,b] is said to be of type 1 if there exists a constant C_1 independent of μ such that for any $x \in [a,b]$

$$|f^{(\mu)}(x)| \leq M_\mu$$

and

$$M_\mu = O(C_1^\mu \mu!) \text{ as } \mu \to \infty$$

It is easy to show that type 1 functions include functions like $1/x$, $\log x$, $\sqrt{x}$ ....

Definition 2

A continuously differentiable function f over an interval [a,b] is said to be of type 2 if there exists a constant C_2 independent of μ such that for any $x \in [a,b]$

$$|f^{(\mu)}(x)| \leq M_\mu$$

and

$$M_\mu = O(C_2^\mu) \text{ as } \mu \to \infty$$

It is easy to show that functions like e^x, cosx, sinx... are type 2 functions.

7.2.2 Memory Complexity of the Method

Lemma 1

Let $\mathcal{M}_1(n)$ be the number of single precision ROM words necessary to evaluate a type 1 function f(x) to precision n, uniformly for all floating-point numbers $x \in [a,b]$ $(\infty < a < b < +\infty)$ using the R-method with polynomials of degree $\mu(n)$. Then as $h \to \infty$ and $\mu(n) \to \infty$,

$$\mathcal{M}_1(n) = O(n\mu 2^{\frac{n}{\mu}}) \tag{7.2}$$

Proof: Using (7.1) we wish to satisfy the condition:

$$\frac{h^{\mu+1} M_{\mu+1}}{2^{\mu}\ (\mu+1)!} \leq C 2^{-n} \tag{7.3}$$

By the definition of a type 1 function,

$$M_{\mu+1} = O(C_1^{\mu}\mu!)$$

We then get

$$\frac{h^{\mu+1} M_{\mu+1}}{2^{\mu}\ (\mu+1)!} \sim \frac{h^{\mu} C_1^{\mu}}{2^{\mu}} \quad \text{as } \mu \to \infty$$

Since $\mu + 1 = O(\mu)$, (7.3) becomes

$$\frac{h^{\mu} C_1^{\mu}}{2^{\mu}} \leq C 2^{-n}$$

so

$$h = O(2^{-\frac{n}{\mu}})$$

The number of addressable ROM words is

$$R_1(n) = \frac{b-a}{h} = O$$

Because $(\mu + 1)$ coefficients are needed for each polynomial this value must be multiplied by $(\mu + 1)$ to obtain the total number of words. In addition, each coefficient has length $O(n)$ and thus

$$\mathcal{M}_1(n) = O(n 2^{\frac{n}{\mu}})$$

Lemma 2

Let $\mathcal{M}_2(n)$ be the number of single precision ROM words necessary to evaluate a type 2 function $f(x)$ to precision n, uniformly, for all floating point numbers $x \in [a,b]$ $(-\infty < a < b < +\infty)$ using the R-method with polynomials of degree $\mu(n)$. Then as $n \to \infty$ and $\mu(n) \to \infty$:

$$\mathcal{M}_2(n) = O(n 2^{\frac{n}{\mu}})$$

Proof: Using (7.1) we wish to satisfy the condition:

$$\frac{h^{\mu+1} M_{\mu+1}}{2^{\mu}\ (\mu+1)!} \leq C2^{-n}$$

By the definition of a type 2 function,

$$M_{\mu} = O(C_2^{\mu}) \text{ as } \mu \to \infty$$

We then get

$$\frac{h^{\mu+1}\ M_{\mu+1}}{2^{\mu}\ (\mu+1)!} \simeq \frac{h^{\mu} C_1^{\mu}}{2^{\mu}\mu!} \qquad \text{as } \mu \to \infty$$

Again $\mu + 1 = O(\mu)$ and, using Stirling's formula,*:

$$\frac{h^{\mu} C_1^{\mu}}{2^{\mu}\mu!} \simeq \frac{h^{\mu}}{2^{\mu}\mu^{\mu}} (C_1 e)^{\mu} (2\pi\mu)^{-\frac{1}{2}}$$

Inequality (7.5) then becomes:

$$\frac{h^{\mu}}{\mu^{\mu}} \frac{C_1^{\mu} e^{\mu}}{2^{\mu}} (2\pi\mu)^{-\frac{1}{2}} < C2^{-n}$$

---

* $n! \sim \left(\frac{n}{e}\right)^n \sqrt{2\pi n}$

Solving for h yields:

$$h \simeq C_3 \mu \cdot \mu^{\frac{1}{2\mu}} \, 2^{-\frac{n}{\mu}}$$

As $\mu \to \infty$, $\mu^{\frac{1}{2\mu}} \to 1$ and therefore the number of addressable ROM words is

$$R_2(n) = \frac{b-a}{h} = O\left(\frac{1}{\mu} \, 2^{\frac{n}{\mu}}\right)$$

Multiplying by the number of polynomial coefficients and the word length of each coefficient:

$$\mathcal{M}_2(n) = O\left(n 2^{\frac{n}{\mu}}\right)$$

To summarize, the number of ROM words for type 1 functions (x, 1/x, logx...) grows μ times faster than the number of ROM words for type 2 functions (e^x, cosx , sinx...) when $\mu, n \to \infty$. The bounds for type 1 and type 2 functions will differ accordingly due to the factor $\mu(n)$.

7.2.3 The Relationship between Polynomial Degree and ROM Size

It would be interesting to know if there is a way to increase $\mu(n)$ as $n \to \infty$ such that the total number of ROM words does not grow "too fast." More specifically:

What $\mu(n)$ will guarantee $\varepsilon_o = 0(2^{-n})$ and maintain $\mathcal{M}_1(n)$ or $\mathcal{M}_2(n) < 0(n^k)$ for some fixed k greater than 0?

We next present two theorems which answer this question. They establish a growth rate for $\mu(n)$ which guarantees a polynomial growth rate for the number of ROM words (single precision).

Theorem 1

If $-\infty<a<b<\infty$,

$$R_1 = O(n^{2+\gamma})$$

single precision Read Only Memory words suffice to evaluate a type 1 function f(x) to precision n, uniformly, for all floating point numbers $x\varepsilon[a,b]$ using the R-method with polynomials of degree

$$\mu_{R_1}(n) = \frac{n}{\gamma\log n+\log(\gamma\log n)+ O(\frac{\log\log n}{\log n})}$$

where γ is an arbitrary positive constant.

This theorem states that the number of ROM words can be kept under $n^{2+\gamma}$ if $\mu(n)$ increases as shown above. Notice that γ cannot be null.

Proof: From Lemma 1

$$\mathcal{M}_1(n) = O(n\cdot 2^{\frac{n}{\mu}})$$

We wish to find a $\mu(n)$ such that:

$$\mu \; n \; 2^{\frac{m}{\mu}} = n^k$$

or

$$\mu \; 2^{\frac{n}{\mu}} = n^{k-1}$$

By taking the log (base 2) of both sides of the equation:

$$\log \mu + \frac{n}{\mu} = (k-1)\log n$$

which can be written:

$$\mu = \frac{n}{(k-1)\log n - \log \mu}$$

To solve this equation we use an iterative method assuming n large:

$$\mu_{i+1} = G(\mu_i)$$

Choosing $\mu_o = n$ one obtains

$$\mu_1 = \frac{n}{(k-2)\log n}$$

Let $k-2 = \gamma$ so that

$$\mu_1 = \frac{n}{\gamma \log n}$$

and $\gamma > 0$ allows μ_1 to remain finite.

The next iteration gives:

$$\mu_2 = \frac{n}{(\gamma+1)\log n - \log\mu_1}$$

replacing μ_1:

$$\mu_2 = \frac{n}{\gamma\log n + \log(\gamma\log n)}$$

The third iteration is: $\mu_3 = G(\mu_2)$

$$\mu_3 = \frac{n}{(\gamma+1)\log n - \log\mu 2}$$

or, replacing μ_2:

$$\mu_3 = \frac{n}{\gamma\log n + \log(\gamma\log n) + \frac{\log(\gamma\log n)}{\text{Log2}.\log n} - \frac{(\log(\gamma\log n))^2}{2\text{Log2}.(\gamma\log n)^2}}$$

which, by using $\ln(1+x) = -\frac{x^2}{2} + \ldots$ becomes

$$\mu_3 = \frac{n}{(\gamma+1)\log n + \log n + \log(\gamma\log n + \log(\gamma\log n))}$$

and therefore

$$\mu_3(n) = \frac{n}{\gamma\log n - \log(\gamma\log n) + 0\left(\frac{\log\log n}{\log n}\right)}$$

Recall that $k = \gamma + 2$ and therefore $n^k = n^{2+\gamma}$ so that $\mathcal{M}_{R1}(n) = 0(n^{2+\gamma})$.

Theorem 2

If $-\infty < a < b < \infty$

$$\mathcal{M}_{R2}(n) = 0(n^{1+\gamma})$$

single precision Read Only Memory words suffice to evaluate type 2 functions $f(x)$ to precision n, uniformly, for all floating point numbers $x \in [a,b]$ using the R-method with polynomials of degree

$$\mu_{R2}(n) = \frac{n}{\gamma\log n}$$

as $n \to \infty$ and where γ is an arbitrary positive constant.*

Proof: From Lemma 2

$$\mathcal{M}_2(n) = 0(n2^{\frac{n}{\mu}})$$

Replacing μ by $\frac{n}{\gamma\log n}$ where $\gamma = k - 1$ yields:

---

*Note that the relationship between γ and k is different than in 1. γ is simply the smallest positive number allowing $\mu(n)$ to remain finite for finite n.

$$\mathcal{M}_{R2}(n) = O(2^{\gamma \log n} \cdot n)$$

or

$$\mathcal{M}_{R2}(n) = O(n^{1+\gamma})$$

In theorem 1 and 2 we were concerned with polynomial growth of the total number of ROM words as a whole. The next two theorems deal with polynomial (n^k) growth of the number of ROM addresses (denoted $R_1(n)$ and $R_2(n)$). Then, in order to obtain the total number of single precision ROM words it is necessary to multiply $R_1(n)$ or $R_2(n)$ by $\mu(n) \cdot n$. The next theorem is presented without proof.

Theorem 1-A

If $-\infty < a < b < \infty$

$$\mathcal{M}_{R1} = \frac{1}{\gamma} O\left(\frac{n^{2+\gamma}}{\log n}\right)$$

single precision Read Only Memory words suffice to evaluate a type 1 function $f(x)$ to precision n, uniformly, for all floating point numbers $X \in [a,b]$ using the R-method with polynomials of degree

$$\mu'_{R1}(n) = \frac{n}{\gamma \log n}$$

as $n \to \infty$ where γ is an arbitrary positive constant. In this case $\gamma = k$.

Theorem 2-A

If $-\infty < a < b < \infty$

$$\mathcal{M}'_{R2} = \frac{n^{\gamma+1}}{\gamma\log n - \log(\gamma\log n) + 0\left(\frac{\log\log n}{\log n}\right)}$$

single precision Read Only Memory words suffice to evaluate a type 2 function f(x) to precision n, uniformly, for all floating point numbers $x \in [a,b]$ using the R-method with polynomial degrees

$$\mu'_{R2}(n) = \frac{n}{\gamma\log n - \log(\gamma\log n) + 1/\gamma O\left(\frac{\log\log n}{\log n}\right)}$$

as $n \to \infty$ where γ is an arbitrary positive constant.

Proof: To maintain the number of addresses bounded by n, one must find a $\mu(n)$ such that:

$$R_2(n) = \frac{2^{\frac{n}{\mu}}}{\mu} = n^k$$

taking the logs of both sides:

$$\frac{n}{\mu} - \log \mu = k\log n$$

and the equation above, put in the form $\mu = G(\mu)$ becomes:

$$\mu = \frac{n}{k\log n+\log\mu}$$

and is solved recursively.

The method is similar to the proof obtained for theorem 1 choosing $\mu_o = n$ as the initial value and replacing gives the first iteration μ_1:

$$\mu_1 = \frac{n}{(k+1)\log n}$$

$\mu_2 = G(\mu_1)$:

$$\mu_2 = \frac{n}{k\log n+\log\mu_1}$$

replacing with μ_1:

$$\mu_2 = \frac{n}{k\log n+\log n-\log((k+1)\log n)}$$

$$\mu_2 = \frac{n}{(k+1)\log n-\log((k+1)\log n)}$$

Continuing this way, the result is after a third iteration and using $\ln(1-x) \simeq -x - \frac{x^2}{2} - \ldots$:

$$\mu = \frac{n}{(k+1)\log n-\log(k+1)\log n+0(\frac{\log\log n}{\log n})}$$

multiplying n^k by $n \cdot \mu$ gives the total memory

$$\mathcal{M}_{R2}(n) = \frac{n^{k+2}}{(k+1)\log n + \log(k+1)\log n + 0\left(\frac{\log\log n}{\log n}\right)}$$

replacing (k+1) by γ gives the result.

7.3 The Effectiveness of the R-Method

7.3.1 The S-Method as a Particular Case of the R-Method

For the parallel polynomial evaluation method, the function f(x) is approximated by a single polynomial over the entire interval [a,b]. The following two lemmas give an order of magnitude of the growth of the degree of the polynomial that approximates f(x) with precision n in [a,b]. It is important to note that the amount of memory will not stay constant. This is simply due to the fact that to "keep up" with the increasing precision, the degree of the polynomial must also increase and therefore the number of polynomial coefficients must increase to infinity.*

*This is due to a well-known theorem of approximation theory established by Kolmogorov and sometimes referred to as the "n-width theorem." A consequence is that it is not possible to find a sequence of polynomials $P_\mu(x)$ of degree μ having a finite number of coefficients, such that $P_\mu(x)$ converges uniformly to f(x) in [a,b] as $n \to \infty$.

Lemma 3

If $-\infty < a < b < \infty$, the S-method can evaluate a type 1 function f(x) to precision n, uniformly, for all floating point numbers x [a,b] as $n \to \infty$ if the degree $\mu_{S1}(n)$ of the polynomial approximant is:

$$\mu'_{S1}(n) = n$$

and therefore the amount of memory is

$$\eta'_{S1}(n) = O(n^2)$$

Proof: For a type 1 function the number of addressable words is $O(2^{\frac{n}{\mu}})$

If $2^{\frac{n}{\mu}}$ is to stay of O(1) (since there is only one polynomial) this implies

$$\frac{n}{\mu} = O(1) \text{ or } \mu = n$$

Each coefficient has word length O(n) and therefore:

$$\eta'_{S1}(n) = O(n^2)$$

Lemma 4

If $-\infty < a < b < \infty$, the S-method can evaluate a type 2 function f(x) to precision n, uniformly, for all floating point numbers $\varepsilon [a,b]$ as $n \to \infty$ with the degree $\mu'_{S2}(n)$ of the polynomial such that:

$$\mu'_{R2}(n) = \frac{n}{\log n - \log\log n + 0\left(\frac{\log\log n}{\log n}\right)}$$

Proof: The proof can be inferred by trying to find such that:

$$\frac{2^{\frac{n}{\mu}}}{\mu} = 1$$

By taking the log:

$$\frac{n}{\mu} - \log \mu = 0$$

or

$$\mu = \frac{n}{\log \mu}$$

Let us fix n and solve this equation for μ. Let us give an initial value to $\mu_o = n$.

$$\mu_1 = \frac{n}{\log n}$$

iterating:

$$\mu_2 = \frac{n}{\log\mu_1} = \frac{n}{\log n - \log\log n}$$

$$= \frac{n}{\log n\left(1 - \frac{\log\log n}{\log n}\right)}$$

iterating again:

$$\mu_3 = \frac{n}{\log\mu_2}$$

$$\mu_3 = \frac{n}{\log n - \log\left(\log n\left(1 - \frac{\log\log n}{\log n}\right)\right)}$$

$$\mu_3 = \frac{n}{\log n - \log\log n - \log\left(1 - \frac{\log\log n}{\log n}\right)}$$

using

$$\ln(1-x) \approx x + \frac{1}{2}x^2 + \ldots$$

and developing the log yields

$$\mu_3 = \frac{n}{\log n - \log(\log n) - \frac{\log(\log n)}{\text{Log2}.\log n} + \frac{(\log(\log n))^2}{2\text{Log2}.(\log n)^2}}$$

which can be written

$$\mu'_{S2}(n) = \frac{n}{\log n - \log(\log n) + O\left(\frac{\log\log n}{\log n}\right)}$$

where

$$\frac{\text{loglogn}}{\text{logn}} \quad \rightarrow 0 \text{ when } n \rightarrow \infty$$

Each coefficient has a word length O(n) and therefore:

$$S2'(n) = \frac{n^2}{\log n - \log \log n + O(\frac{\log \log n}{\log n})}$$

Lemmas 3 and 4 give an idea of how fast the degree of the polynomial which approximates f(x) over [a,b] must grow in order to guarantee a precision O(n). In essence, a type 2 function can be approximated by a polynomial having a degree smaller by a factor of (logn - loglogn) than that required for a type 1 function approximated with the same precision. It can be seen from theorem 1 that for type 2 functions the S-method is a particular case of the R-method where the number of ROM addresses is $O(n^{\gamma-1})$ with $\gamma = 1$.

7.3.2 The Number of Processors, Speed-Up, Cost, and Efficiency for the R-Method

The number of processors for the R-method can be finite and independent of p, or can be equal to 2μ if Maruyama's algorithm is used. In what follows, however, we

will assume that only one multiplier is available after the bank of ROMs so that we can distinguish the effect of the memory approach from that using a parallel polynomial evaluation scheme. We assume that Horner's Rule is used for the evaluation of $P_{\mu}(x)$.

We can now easily obtain, as a function of μ and n, the various measures as defined by Kuck for type 1 and type 2 functions. We will use indexes 1 and 2 for the two types of functions. In all of the derivations it is assumed that the address delay is negligible compared to the polynomial evaluation time.

Number of Processors

The number of processors will be the number of single precision memory words.

Speed-up:

T_S is the number of steps using one serial processor (that is: using one multiplier and Horner's rule).

From Lemma 3, $T_{S1} = \mu'_{S1}(n) = n$

From Lemma 4, $T_{S2} = \mu'_{S2}(n) = \frac{n}{\log n - \log\log n}$

From Theorem I or I-A, $T_{R1} = \mu_{R1}(n)$ or $\mu'_{R1}(n)$

From Theorem II or II-A $\quad T_{R2} = \mu_{R2}(n)$ or $\mu'_{R2}(n)$

Then

$$S_{R1} = \frac{n}{\mu_1} \quad \text{and} \quad S_{R2} = \frac{n}{\mu_2(\log n - \log\log n)}$$

where μ_1 can be μ_{R1} or μ'_{R1} and μ_2 can be μ_{R2} or μ_{R2}'.

If, for the R-method, we use the scheme that limits the growth of the number of addresses to n^k (i.e., $\mu'_{R1}(n)$ and $\mu'_{R2}(n)$)

$$S_{R1} = \gamma \log n \qquad S_{R2} = \gamma \quad \text{(for large n)}$$

Cost:

The cost is the product p . T and we derive easily:

$$C_{R1} = P_{R1}\, T_{R1} = n\, \mu^2\, 2^{n/\mu}$$

$$C_{R2} = P_{R2}\, T_{R2} = n\, \mu\, 2^{n/\mu}$$

where $P_{R1} = \mathcal{M}(n)$ and $P_{R2} = \mathcal{M}_2(n)$ since the number of processors is equal to the number of single precision memory words.

Effectiveness:

$$E_{R1} = \frac{S_{R1}}{C_{R1}} = \frac{1}{\mu_1^3 \, 2^{n/\mu}}$$

$$E_{R2} = \frac{S_{R2}}{C_{R2}} = \frac{1}{\mu_1^2 (\log n - \log\log n) \, 2^{n/\mu}}$$

If we use a scheme that limits R(n) to n^k (i.e., the address limited case) we obtain:

$$E_{R1} = \frac{\gamma^3 \, (\log n)^3}{n^{\gamma+3}}$$

$$E_{R2} = \frac{\gamma^3 \, (\log n)^2}{n^{\gamma+2}} \qquad \text{(for large n)}$$

7.3.3 Comparisons Using 1 Processor

We will assume that the evaluation of the polynomials will be done using Horner's Rule with one processor. We want to express the ratio:

$$\eta = \frac{S_R}{C_R} \Big/ \frac{S_S}{C_S}$$

as a function of μ and n. We are not choosing the polynomial degree $\mu(n)$ a priori but rather asking ourselves the question:

Given n, relatively large, and the task of designing a function generator, can we find a function of $\mu(n)$ such that the ratio of effectiveness is greater than or equal to 1? We will therefore compute η for type 1 and type 2 functions and determine $\mu(n)$ such that $\eta \geq 1$.

Type 1 functions:

$$\eta_1 = \frac{P_{S1}}{P_{R1}} \cdot \frac{T^2_{S1}}{T^2_{R1}}$$

We have:

$$P_{S1} = \mathcal{P}'_{S1}(n) = n^2$$

and

$$P_{R1} = \mathcal{P}'_{R1}(n) = n \mu 2^{n/\mu}$$

$$T_{S1} = n$$

and

$$T_{R1} = \mu$$

Thus

$$\eta_1 = \frac{n^2}{n \mu 2^{n/\mu}} \quad \frac{n^2}{\mu^2}$$

$$\eta_1 = \frac{(\frac{n}{\mu})^3}{2^{n/\mu}}$$

and we want to determine $\mu(n)$ such that

$$\eta_1 > 1$$

We can study the function

$$\eta_1(x) = \frac{x^3}{2^x}$$

and it is easy to find that $\eta_1(x)$ has a maximum at $x = 4.328$ ... equal to 4.036 and that the equation $\eta_1(x) = 1$ has two roots, one at $x = 1.368...$ and one at $x = 9.937....$

$$\mu(n) = \frac{n}{4.036} = 6(n)$$

will be the most effective choice of the polynomial degree.

For $\mu = 6(n)$ it is then possible to find a multiplier contrast such that $\eta_1(x)$ is greater than 1 and the R-method is more efficient than the method using one polynomial. This tells us that the optimum number of ROM addresses is, for type 1 functions:

$$R_1 = 6(1)$$

Type 2 Functions:

$$\eta_2 = \frac{P_{S2}}{P_{R2}} \frac{T^2_{S2}}{T^2_{R2}}$$

We have:

$$P_{S2} = \quad_{S2}(n) = \frac{n^2}{\log n - \log\log n}$$

and

$$P_{R2} = \quad_2 \quad = n\ 2\ \frac{n}{\mu}$$

$$T_{S2} = \frac{n}{\log n - \log\log n}$$

and

$$T_{R2} = \mu$$

Therefore:

$$\eta_2 = \frac{n^3}{2\frac{n}{\mu}\ \mu^2(\log n - \log\log n)^3}$$

and we want to find $\mu(n)$ such that $\eta_2 \geq 1$.

To study this function we first make the change of variable:

$$x = 2\ \frac{n}{\mu}$$

and η_2 becomes

$$\eta_2 = \frac{(\log x)^2}{x} \cdot \frac{n}{(\log n - \log\log n)^3}$$

We are looking for x such that:

$$\frac{(\log x)^2}{x} > \frac{(\log n - \log n)^3}{n}$$

One can show that x is approximated by:

$$x \leqslant \frac{n}{\log n}\left(1 - O\left(\frac{\log\log n}{\log n}\right)\right)$$

and therefore

$$\mu \geqslant \frac{n}{\log n - \log\log n + O\left(\frac{\log\log n}{\log n}\right)}$$

will satisfy $\eta_2 > 1$. This is the same expression for μ as is found in Lemma 4.

Consequently, asymptotically the choice of μ corresponds to the S method and, for type 2 functions the R-method is asymptotically equivalent to the S-method for the same efficiency.

7.3.4 Comparisons Using p Processors

From Maruyama's bound, the S-method for type 1 functions gives:

$$T_S = O(\log\mu) = O(\log n)$$

For the R-method

$$T_R = O(\log\mu(n))$$

If the S-method is used to evaluate f(x) with precision n, the total hardware requirements are:*

Amount of Memory + Number of Multipliers (Processors)

From Maruyama's bound, the number of multipliers is $2\mu p(n) = 2\,O(n) = O(n)$. From Lemma 3 the amount of memory is $O(n^2)$.

So for the S-method

$$p_S = O(n^2) + O(n) = O(n^2)$$

If the R-method is used, from Lemma 1

$$\text{Amount of Memory} = O(n\mu 2^{\frac{n}{\mu}})$$

$$\text{Number of processors} = O(\mu(n))$$

so

$$p_R = O(\mu(1+ n\, 2^{\frac{n}{\mu}}))$$

The ratio η is then:

$$\eta = \frac{n^2}{\mu(1+n2^{n/\mu})} \frac{(\log n)^2}{(\log\mu)^2}$$

*Recall that we are counting single precision multipliers, that is the size of a multiplier is 6(1). In the same way we count single precision memory words.

Is there a $\mu(n)$ such that $\eta > 1$? This would imply that effectiveness is gained by using some Read Only Memory.

Again we must study the function $\eta(x)$ using $x = \frac{n}{\mu}$:

$$\eta(x) = \frac{x}{2^x} \frac{1}{\left(1 - \frac{\log x}{\log n}\right)^2}$$

We want

$$\frac{x}{2^x} \geqslant \left(1 - \frac{\log x}{\log n}\right)^2$$

which implies:

$$1 - \frac{\sqrt{x}}{2^{\frac{x}{2}}} \leqslant \frac{\log x}{\log n}$$

The left number is equal to .3 for $x = 1$ and $x = 2$ and has a minimum between 1 and 2.

This inequality always has a solution, and because μ must grow to infinity with n, polynomial degrees μ exist and are such that:

$$\mu \leq \frac{n}{x(n)}$$

7.3.5 Comparisons to the Best Known Bound for Function Evaluation

The best known bound for function evaluation has been determined by Brent [BRE74] [BRE76]. He showed, by using ascending Landen transformations and elliptic integrals, that the elementary functions can be evaluated in:

$$0(\log n\ \gamma(n))\ \text{time}$$

where $\gamma(n)$ is the Schönhage/Strassen bound. This assumes a single processor and $0(n)$ memory.

For the R-method, by evaluating polynomials in parallel in less than $2 \log \mu$ time and by choosing a degree $\mu(n)$ such that

$$\mu(n) = \frac{}{\gamma \log n}$$

with memory bounded by $n^{\gamma+2}$, the speed achievable is

$$0(\log \mu \cdot \gamma(n)) = 0 \log \left(\frac{n}{\gamma \log n}\right) \gamma(n)$$

which is asymptotically equivalent to the bound given by Brent. It is a slight improvement of the Brent's bound at the expense of $0(n^{\gamma+1})$ words of memory and $2 \frac{n}{\gamma \log n}$ processors.

8. CONCLUSIONS

8.1 Areas of Further Research

Curves of Average Speed vs Number of ROM Words

A detailed study of the curves representing the average number of multiplications vs the number of ROM words for the most common functions and for word lengths up to, say, 120 bits should be made. The curves would be of interest in determining the feasibility of a particular implementation given the design constraints. This study should be done for the 3 cases where:

- only one multiplier is available
- a finite number of multipliers, P, is available
- an unlimited number of multipliers can be used in parallel to evaluate the polynomial.

In previous chapters the curves represented a lower bound due to the fact that digitization was not taken into account. It would be useful to have more data on the quantitative effects of digitization on the curves.

Polynomial Approximants

We assumed the most general Chebyshev polynomial in our developments. But in practice, the polynomials can be chosen with an even or odd degree, depending on the function,

or can be economized using telescoping series [BEL68]. This would further contribute to a reduction in memory.

Finding the Joints Simultaneously for Several Functions

To find the position of the joints for a function $f_1(x)$, we have built a function $Q_1(\underline{X},\mu)$ whose minimization gave a solution for the vector of joint locations $\underline{X}$, for the NLP problem as defined in Section 3.

When the hardware implementation of the algorithm for $f_1(x)$ was considered, we saw that a Read Only Memory decoder was needed in order to detect the joints closest to the argument x and thus determine the number of ROM accesses (i.e., number of multiplications) necessary to evaluate $P_{m_j}^{(i)}(x)$.

Given another function $f_2(x)$ to be implemented, the positions of the joints for this function would likely be entirely different from the positions of the joints for $f_1(x)$, and consequently, the same bank of ROMs could not be used for address decoding and joint interval determination for both $f_1(x)$ and $f_2(x)$. Would it be feasible to have the same joints for $f_1(x)$ and $f_2(x)$? Stated in a more general way, the question raised is, can one solve several of these NLP problems simultaneously. We build a function $Q_1(\underline{X},\mu)$ for function $f_1(x)$, a function $Q_2(\underline{X},\mu)$ for function $f_2(x)$, etc.... and we try to solve for $\underline{X}$:

$$\begin{cases} \text{minimize } Q_1(\underline{X},\mu) \\ \text{minimize } Q_2(\underline{X},\mu) \\ \quad \vdots \\ \text{minimize } Q_p(\underline{X},\mu) \quad \text{simultaneously.} \end{cases} \tag{8.1}$$

Fig. 8.1 illustrates this problem for the three functions:

$$f_1(x) = 1/x \text{ in } [.5, 1]$$

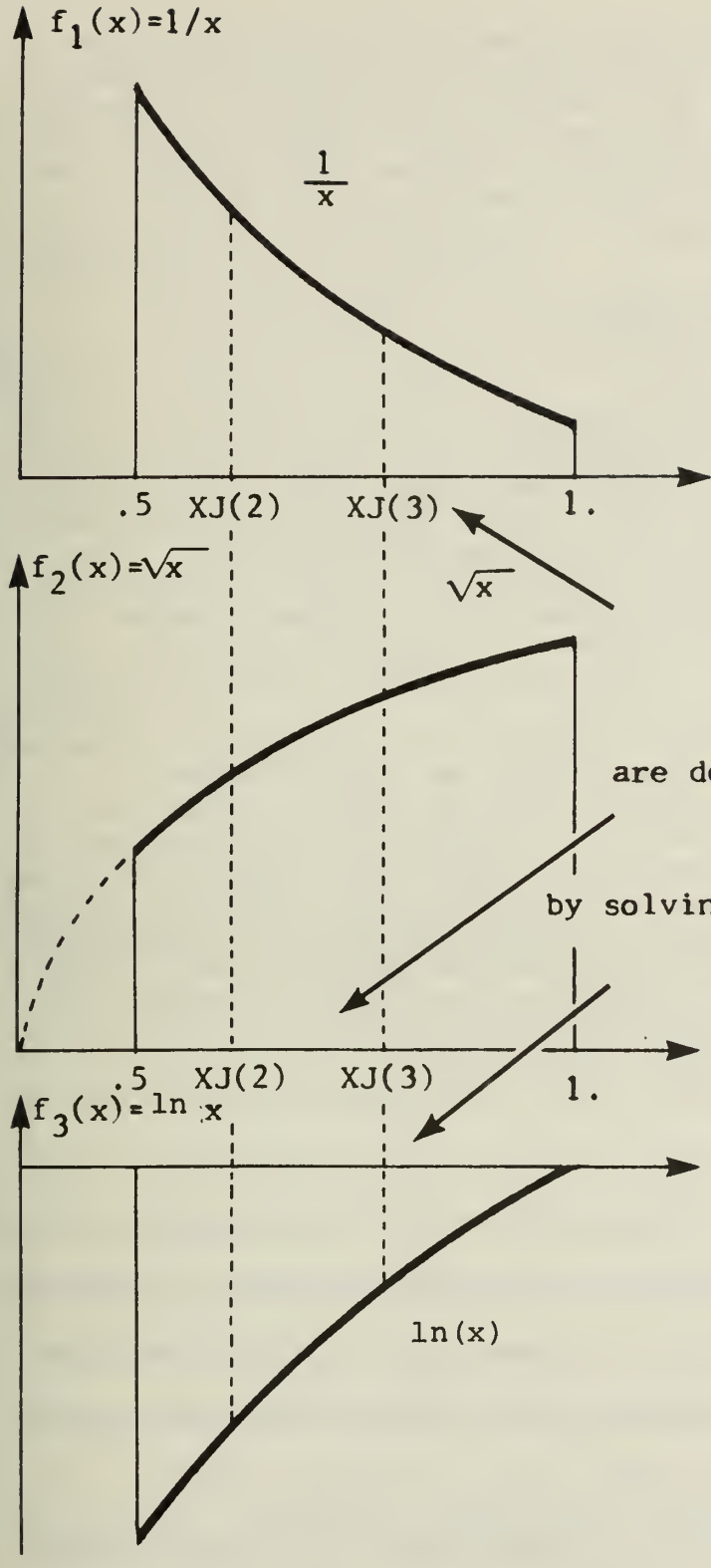
$$f_2(x) = \sqrt{x} \text{ in } [.5, 1]$$

$$f_3(x) = \ln x \text{ in } [.5, 1]$$

More research in this direction would be useful in the sense that having a unique set of joints for several functions would simplify considerably the structure of the arithmetic unit. The difficulty lies in minimizing several functions at once. One possibility would be to replace the general problem (8.1) by the simpler problem:

$$\text{minimize } (\lambda_1 Q_1(\underline{X},\mu) + \lambda_2 Q_2(\underline{X},\mu) + \ldots + \lambda_p Q_p(X,\mu))$$

where the constants λ_i could be chosen proportionally to the frequency of calls to each function $f_i(x)$ in a representative set of scientific programs, for example.



The positions of the Joints XJ(2) and XJ(3) are determined simultaneously for the 3 functions by solving a general NLP problem

Figure 8.1. Optimal Determination of the Joints for Several Functions Simultaneously

Care must be taken, because the solution of the general problem (8.1) by a method like Newton's or steepest descent may involve non-square matrices and overdetermined or underdetermined systems: the number of joints is (s-1) and the number of functions to be minimized is P, and, in general $P \neq s-1$. We would like to point out that a technique such as the generalized inverse* may prove useful to solve this kind of problem. (IMSL has a subroutine to compute the generalized inverse of a rectangular matrix.). The effect of a global minimization on the average number of multiplications of each individual function would have to be determined. The whole question of the relative size and growth rates of the address decoding memory versus the coefficient memory needs to be considered. Which one grows faster? The total amount of memory needs to be considered, particularly in a case where the same joint locations are being forced onto several functions.

Theoretical Issues

As mentioned in Section 7, the theorem stating that "a deterministic multitape Turing machine of time complexity t(n) could be simulated by a deterministic Turing machine of tape complexity t(n)/logt(n)" is related to the problem

*Sometimes called "the pseudo inverse." It allows solving a system of the form AX=b where A is a rectangular or even singular matrix. The solution has a minimum norm in a least-square sense, in the case of Moore-Penrose inverse.

of replacing some of the computations by lookup. What the exact relationship is, and what this theorem tells us about the limits and capabilities of the R-method remains to be established in a formal way.

The behavior of the multiplicative constants involved in the derivations of Section 7 could be fruitfully investigated. Using this, the determination of the value of n where Brent's algorithm using ascending Landen transformations becomes faster and/or more efficient than the R-method for a given technology and amount of hardware, could be derived.

Bounds should be derived assuming that the size of the multiplier is allowed to grow with n and is implemented with Read Only Memory which allows evaluation in $O(\log n)$ time. In our derivations we have assumed that a multiplier of fixed precision was used. A bound is needed on the time to multiply two n-bit integers with precision $O(n)$* as well as on the amount of ROM needed to implement such a multiplier. As memory becomes cheaper and cheaper, it will be possible to design generalized monomial or polynomial evaluators. For example, special purpose hardware using ROMs could be designed to evaluate terms of the form ax^n. Further research on the bound to evaluate ax^n in hardware should be considered.

---

*Bounds already exist if the precision is $O(n^2)$. (Kuck)

Finally, the question raised in Section 5 of the possible lattice structure of the various implementations with the partial ordering relationship "to have an implementation always faster for the same amount of ROM," should be investigated in more detail.

8.2 Summary and Conclusions

We have presented a method for function generation which emphasizes the capabilities of LSI by using large-scale Read Only Memory in conjunction with a parallel multiplier as the basic operator, for evaluating polynomials. The design of the function generator is presented as a trade-off between speed of evaluation, expressed in terms of average number of multiplications, and cost, in terms of number of Read Only Memory words available. It has been shown that the minimization of the Mean Running Time, with a constraint on the number of ROM words, is a Non-Linear Programming Problem. The solution yields the optimal location of the joints that separate the partitions within which the degree of polynomial approximant is the same. Within each partition there is a large number of subintervals of small size. The polynomial approximating f(x) in each subinterval has a low degree and therefore requires only a few multiplications for evaluation. The average speed is guaranteed to be optimal by the solution of the NLP program. The coefficients of all of the polynomials are

stored in ROMs. Decoding the address of the coefficient from the argument x presents practical design problems in the VIVD case. The UIVD technique seems the easiest to implement: it has the flexibility of a variable average number of multiplications and the ease of address-decoding with uniform intervals. It has been shown that a speed-up of $\gamma \log n$, or $(\gamma + 1)$ depending on the type of function, is achievable and that the amount of memory needed to achieve this speed-up does not grow exponentially but in polynomial space: n^{γ}.

This method could have a wide span of applications ranging from function generation in microprocessors to high-speed high precision arithmetic units. The structure of the arithmetic unit is conceptually simple and relies on Read Only Memory. The control is straightforward and consists essentially of applying Horner's Rule. The method does not require a dedicated multiplier. More work is needed in order to obtain data for a common arithmetic unit for a class of functions and more data is needed on the amount of ROM vs. speed for large word sizes.

LIST OF REFERENCES

AND67 Anderson, Earle, Goldschmidt and Powers, "Model 91 Floating-Point Execution," IBM Journal, 1967, pp. 34-53.

BAK73 Baker, P.W., "Predictive Algorithms for Some Elementary Functions in Radix 2," Electronic Letters, Vol. 9, No. 21, pp. 493-494, October, 1973.

BAU73 Baugh, Charles R. and Bruce A. Wooley, "A Two's Complement Parallel Array Multiplication Algorithm," IEEE Transactions on Computers, Vol. C-22, No. 12, pp. 1045-47, December, 1973.

BEL68 Bell, Richard A. and S.M. Shah, "Oscillating Polynomials and Approximations to Fractional Powers of x," Journal of Approximation Theory, 1, pp. 269-274, 1968.

BRE74 Brent, Richard P., "The Complexity of Multiple-Precision Evaluation of Elementary Functions," first draft, 48 pp., November, 1974.

BRE76 _____, "Fast Multiple-Precision Evaluation of Elementary Functions," Journal of the Association for Computing Machinery, Vol. 23, No. 2, pp. 242-241, April, 1976.

BRU73 Breuer, David R., "A High-Speed Monolithic 8x8 Multiplier," unpublished document.

CHE66 Cheney, E.W., Introduction to Approximation Theory, International Series in Pure and Applied Mathematics, 259 pp., McGraw-Hill Book Co., 1966.

CHE74 Cheney, E.W. and T.J. Rivlin, "Some Polynomial Approximation Operators," Center for Numerical Analysis, The University of Texas at Austin, 16 pp., November, 1974.

CHI70 Chin, Tung and Algirdas Avizienis, "Combinational Arithmetic Systems for the Approximation of Functions," Spring Joint Computer Conference, pp. 95-107, 1970.

CHU74 Chung, T.J. and S.D. Bedrosian, "Fast Digital Multiplier Based on Iterative Cellular Arrays of ROMs," unpublished document.

DAD65 Dadda, L., "Some Schemes for Parallel Multipliers," Alta Frequenza, Vol. XXIV, No. 5, 1965.

DEL70 DeLugish, Bruce Gene, "A Class of Algorithms for Automatic Evaluation of Certain Elementary Functions in a Binary Computer," Ph.D. Thesis, Department of Computer Science, University of Illinois, Urbana, Illinois, June, 1970.

ERC75 Ercegovac, Milos D., "A General Method for Evaluation of Functions and Computations in a Digital Computer," Ph.D. Thesis, Department of Computer Science, University of Illinois, Urbana, Illinois, August, 1975.

FER67 Ferrari, Domenico, "A Division Method Using a Parallel Multiplier," IEEE Transactions on Electronic Computers, pp. 224-26, April, 1967.

FLE66 Flehinger, B.J., "On the Probability That a Random Integer Has Initial Digit A," American Mathematical Monthly, pp. 1056-1061, December, 1966.

FLY70 Flynn, Michael J., "On Division by Functional Iteration," IEEE Transactions on Computers, Vol. C-19, No. 8, pp. 702-6, August, 1970.

HAB70 Habibi, A. and P.A. Wintz, "Fast Multipliers," IEEE Transactions on Computers, pp. 153-157, February, 1970.

HAM70 Hamming, R.W., "On the Distribution of Numbers," The Bell System Technical Journal, Vol. 49, No. 8, pp. 1609-1625, October, 1970.

HAR68 Hart, J.F., Computer Approximations, New York, John Wiley and Sons, Inc., 1968.

HO73 Ho, Irving T. and Tien Chi Chen, "Multiple Addition by Residue Threshold Functions and Their Representation by Array Logic," IEEE Transactions on Computers, Vol. C-22, No. 8, pp. 762-767, August, 1973.

HOP77 Hopcroft, John, Wolfgang Paul and Leslie Valiant, "On Time Versus Space," Journal of the Association for Computing Machinery, Vol. 24, No. 2, pp. 332-337, April, 1977.

IBM68 System/360 Scientific Subroutine Package (360A-CM-03X) Vesion III Programmer's Manual.

IRW77 Irwin, Mary Jane, "An Arithmetic Unit for On-Line Computation," Ph.D. Thesis, Department of Computer Science, University of Illinois, Urbana, Illinois, June, 1977.

JOH73 Johnson, N., "Improved Binary Multiplication System," Electronics Letters, Vol. 9, No. 1, pp. 6-7, January, 1973.

KAN73 Kaneko, Toyohisa and Bede Liu, "On Local Roundoff Errors in Floating-Point Arithmetic," Journal of the Association for Computing Machinery, Vol. 20, No. 3, pp. 391-8, July, 1973.

KIN71 Kingsbury, N.G., "High-Speed Binary Multiplier," Electronics Letters, Vol. 7, No. 10, pp. 277-78, May, 1971.

KIO75 Kiostelidis, John B. and John K. Petrou, "A Piece-wise Linear Approximation of $\log_2 x$ with Equal Maximum Errors in All Intervals," IEEE Transactions on Computers, Vol. C-24, No. 9, pp. 858-860, September, 1975.

KUC74 Kuck, D.J., "On the speed up and cost of Parallel Computation," Proceedings on the Computational Complexity of Problem Solving, The Australian National University, December, 1974.

LAK76 Lakshmi, Goyal, "A Study in the Design of an Arithmetic Element for Serial Processing in an Iterative Structure," Ph.D. Thesis, Department of Computer Science, University of Illinois, Urbana, Illinois, 1976.

LIN70 Ling, H., "High-Speed Computer Multiplication Using a Multiple-Bit Decoding Algorithy," IEEE Transactions on Computers, Vol. C-19, No. 8, pp. 706-709, August, 1970.

LUT65 Luttmann, F.W. and T.J. Rivlin, "Some Numerical Experiments in the Theory of Polynomial Interpolation," IBM Journal, pp. 187-191, May, 1965.

MAR73 Maruyama, K., "On the Parallel Evaluation of Polynomials," IEEE Transactions on Computers, Vol. C-22, pp. 2-5, January, 1973.

MUN73 Munro, I., and Paterson, M., Optimal algorithm for parallel polynomial evaluation. JCSS 7 (1973), 189-198.

PEZ71 Pezaris, Stylianos D., "A 40-ns 17-Bit by 17-Bit Array Multiplier," IEEE Transactions on Computers, pp. 442-447, April, 1971.

RIV74 Rivlin, Theodore J., The Chebyshev Polynomials, John Wiley and Sons, New York, 186 pp. 1974.

ROB76 Robertson, J.E., "Class Notes on Computer Arithmetic," Department of Computer Science, Urbana, Illinois, Fall, 1976.

SCH71 Schonhage, A., and Strassen, V., "Schnelle Multiplikation grosser Zahlen," Computing 7, pp. 281-292, 1971.

SIN73 Singh, Shanker and Ronald Waxman, "Multiple Operand Addition and Multiplication," IEEE Transactions on Computers, Vol. C-22, No. 2, pp. 113-120, February, 1973.

SNI77 Snir, Marc and Amnon B. Barak, "A Direct Approach to the Parallel Evaluation of Rational Expressions with a Small Number of Processors," IEEE Transactions on Computers, Vol. C-26, No. 10, pp. 933-937, October, 1977.

STE72 Stefanelli, R., "Binary Read-Only-Memory Dividers," Unpublished correspondence, 1972.

STE77 Stenzel, Wm. J., Wm. J. Kubitz and Gilles H. Garcia, "A Compact High-Speed Parallel Multiplication Scheme," IEEE Transactions on Computers, Vol. C-26, No. 10, pp. 948-957, October, 1977.

SWE65 Sweeny, D.W., "An Analysis of Floating-Point Addition," IBM Systems Journal, Vol. 4, No. 1, 1965.

SZE59 Szego, Gabor, Orthogonal Polynomials, American Mathematical Society Colloquium Pulbications, Volume XIII, American Mathematical Society, New York, 1959.

TOD63 Todd, John, Introduction to the Constructive Theory of Functions, Birkhaeuser Verlag, Basel und Stuttgart, 127 pp., 1963.

TOR75 Torrero, Edward A., "Focus on Semiconductor Memories," Electornic Design, 7, pp. 98-107, April, 1975.

TSA74 Tsao, Nai-Kuan, "Pn the Distributions of Significant Digits and Roundoff Errors," Communications of the ACM, Vol. 17, Number 5, pp. 269-271, May, 1974.

WAL64 Wallace, C.S., "A Suggestion for a Fast Multiplier," IEEE Transactions on Electronic Computers, pp. 14-17, February, 1964.

APPENDIX

A.1 Program Listings

```
      DOUBLE PRECISION FUNCTION QTOMIN(XMU,NROK,IUNIF,NJOINT,XJ,JDEGR,
     1 BASE,PI,FCT,W,ETA,EPS0)
      IMPLICIT REAL *8 (A-H,O-Z)
      REAL *8 XJ(10)
      EXTERNAL FCT,W,PI
      INTEGER JDEGR(10)
C
C                              THE FUNCTION TO MINIMIZE IS A COMBINATION
C                              OF THE OBJECTIVE FUNCTION TO MINIMIZE THE
C                              NUMBER OF MULTIPLICATIONS AND THE PENALTY
C                              FUNCTION. SEE LUENBERGER PP. 278...
C
      AVMUL1=AVMULT(NJOINT,XJ,JDEGR,BASE)
      PENALT=PENLTY(NROK,IUNIF,NJOINT,XJ,JDEGR,PI,FCT,W,ETA,EPS0)
C                              THIS IS THE FINAL FUNCTION
C
      QTOMIN=AVMUL1+XMU*PENALT
C
C                              THE CONSTANT XMU IS THE PARAMETER WHOSE
C                              VALUE TENDS TO INFINITY .
C                              THE VARIABLES ARE THE JOINTS XJ(I).
C
      RETURN
      END

      DOUBLE PRECISION FUNCTION PENLTY(NROK,IUNIF,NJOINT,
     1 XJ,JDEGR,PI,FCT,W,ETA,EPS0)
      IMPLICIT REAL*8 (A-H,O-Z)
      EXTERNAL FCT,W,PI
      REAL*8 XJ(10)
      INTEGER JDEGR(10)
C
C                              THIS THE PENALTY FUNCTION . SEE LUENBERGER
C                              CHAPTER 12 PP 277...
C                              IN THE PENALTY FUNCTION IS INCLUDED THE
C                               CONSTRAINT NROMWORDS > NROK AND ALSO
C                              THE CONSTRAINT THAT THE SOLUTION HAS TO
C                              VERIFY XJ(1) < XJ(2) <.. <XJ(NJOINT+1).
```

```
C                                XJ(1) AND XJ(NJOIN+1) ARE THE BOUNDS OF THE
C                                X RANGE
C                                THE FOLLOWING PENALTY COMES FROM THE
C                                NUMBER OF ROM WORDS.(< NROK )
C
       ONEK=1024.0D0
      ROMKDF=(NROM(IUNIF,NJOINT,XJ,JDEGR,PI,FCT,W,ETA,EPS0)-NROK)/ONEK
      PEN1=DMAX1(ROMKDF,0.D0)
      PEN1=PEN1*PEN1
      PEN2=0.D0
      DO 10 I=1,NJOINT
      XJDF=XJ(I)-XJ(I+1)
      XM=DMAX1(0.D0,XJDF)
      PEN2=PEN2+XM*XM
10    CONTINUE
      PENLTY=PEN1+ PEN2
9000  RETURN
      END
```

```
      DOUBLE PRECISION FUNCTION AVMULT(NJOINT,XJ,JDEGR,BASE)
      IMPLICIT REAL*8(A-H,O-Z)
      IF(BASE.LT.1.D-16)BASE=2.D0
      REAL*8 XJ(10)
      INTEGER JDEGR(10)
C
C                              THIS FUNCTION COMPUTES THE AVERAGE
C                              NUMBER OF MULTIPLICATIONS
C                              XJ(1),XJ(2),... ,XJ(NJOINT+1) ARE THE
C                              AESCISSAS OF THE JOINTS.
C
C                              JDEGR(1) IS THE DGREE OF THE POLY-
C                              NOMIALS APPROXIMANT IN THE INTERVAL :
C                                        XJ(1) TO XJ(2)
C                              JDEGR(I) IS THE DEGREE OF THE POLYNOMIALS
C                              APPROXIMANT IN THE INTERVAL :
C                                        XJ(I) TO XJ(I+1) ..
C                              NJOINT IS THE NUMBER OF JOINTS INTERVALS.
C                              BASE IS THE BASE REPRESENTATION OF THE
C                              NUMBERS IN THE MACHINE
C
C             COMPUTE AV # OF MULTIPLICATIONS:
C
      X=1.D0
      DO 10 K=1,NJOINT
10    X=X * (XJ(K+1)/XJ(K))**(JDEGR(K))
      AVMULT=DLOG(X)/DLOG(2.D0)
9000  RETURN
      END
```

```
      INTEGER FUNCTION NROM(IUNIF,NJOINT,XJ,JDEGR,PI,FCT
     1,W,ETA,EPS0)
      IMPLICIT REAL*8(A-H,O-Z)
      INTEGER JDEGR(10)
      EXTERNAL PI,W,FCT
      REAL*8 XJ(10)
      NROM=0
      DO 10 I=1,NJOINT
      X1=XJ(I)
      X2=XJ(I+1)
      M=JDEGR(I)
C
C                          THE NUMBER OF ROM WORDS IS EQUAL TO
C                          THE NUMBER OF BREAKPOINTS TIMES THE
C                          DEGREE OF THE POLYNOMIAL +1 .
C
      NROM=NROM+(M+1)*N(IUNIF,X1,X2,M,PI,FCT,W,ETA,EPS0,IER)
10    CONTINUE
C                          COMPUTE THE AMOUNT OF ROM IN K WORDS.
9000      RETURN
      END
```

```
      INTEGER FUNCTION N(IUNIF,X1,X2,M,PI,FCT,W,ETA,EPS0,IER)
C
C                              THIS FUNCTION REPRESENTS THE NUMBER OF
C                              BREAKPOINTS BETWEEN X1 AND X2 FOR POLY-
C                              NOMIAL OF DEGREE M
C
      IMPLICIT REAL*8(A-H,O-Z)
      EXTERNAL PI,FCT,W
       EPS=1.D-10
      IEND=30
      N=1
      IER=0
       IF(DABS(X1-X2).LT.1.D-10)RETURN
      IF(X1.LE.X2)GO TO 9
      XT=X2
      X2=X1
      X1=XT
9     CONTINUE
      XI=X1
10    CALL HINIT(H,XSTRT,XI,M,PI,FCT,W,ETA,EPS0)
         ND=DLOG(1.D0/H)/DLOG(2.D0)
          ND1=ND+1
          H=2.D0**(-ND1)
      IF(ETA.LT.1.D-16)XI=XI+H
      IF(ETA.LT.1.D-16)GO TO 20
      CALL SOLVEQ(U,F,XSTRT,EPS,IEND,IER,XI,M,PI,FCT,W,ETA,EPS0)
      CALL UTOH(U,H,M,EPS0)
         ND=DLOG(1.D0/H)/DLOG(2.D0)
          ND1=ND+1
          H=2.D0**(-ND1)
      XI=XI+H
20    IF(IER.GT.0)GO TO 9000
      IF(XI.GE.X2)GO TO 9000
      N=N+1
      IF(IUNIF.NE.1)GO TO 10
      N=(X2-X1)/H
9000  RETURN
      END
```

```
      SUBROUTINE SOLVEQ(X,F,XSTRT,EPS,IEND,IER,
     1XI,M,PI,FCT,W,ETA,EPS0)
C
C                              SOLVES THE EQUATION FBIG=FSMALL , AFTER
C                              A CHANGE OF VARIABLE U=H**(M+2)/EPS0
C                              X IS THE SOLUTION FOR U
C                              XSTRT IS AN INITIAL VALUE OF U
C                              F IS THE VALUE OF (FBIG-FSMALL) AT THE END
C                              OF THE ITERATION.
C                              THIS ALGORITHM IS A MDIFICATION OF IBM
C                              SS NEWTON'S
C                               FSOL IS THE SUBROUTINE THAT COMPUTES
C                               FBIG-FSMALL AND DERIV(FBIG-FSMALL)
C
      IMPLICIT REAL*8 (A-H,O-Z)
      EXTERNAL PI,W,FCT
      IER=0
      X=XSTRT
      TOL=X
      CALL FSOL(TOL,F,DERF,XI,M,PI,FCT,W,ETA,EPS0)
      TOLF=100.D0*EPS
C
C                              START ITERATION LOOP
C
      DO 6 I=1,IEND
      IF(DABS(F).LT.1.D-6)GO TO 7
C
C                              EQUATION IS NOT SATISFIED BY X
C
1     IF(DABS(DERF).LT.1.D-6)GO TO 8
C
C                              ITERATION POSSIBLE
C
2     DX=F/DERF
      X=X-DX
      TOL=X
      CALL FSOL(TOL,F,DERF,XI,M,PI,FCT,W,ETA,EPS0)
```

```
C
C                                   TEST ON SATISFACTORY ACCURACY
C
      TOL=EPS
      A=DABS(X)
      A1=A-1.D0
      IF(A1.LE.1.D-6)GO TO 4
3     TOL=TOL*A
4     IF((DABS(DX)-TOL).GT.1.D-6)GO TO 6
5     IF((DABS(F)-TOLF).LE.1.0D-6)GO TO 7
6     CONTINUE
C
C                                   END OF ITERATION LOOP
C
C                                   NO CONVERGENCE AFTER IEEND ITERATION STEPS
C                                   ERROR RETURN
      IER=1
7     RETURN
C
C                                    ERROR RETURN IN CASE OF ZERO DIVISOR
C
8      IER=2
9000  RETURN
      END


      SUBROUTINE UTOH(U,H,M,EPS0)
      IMPLICIT REAL*8(A-H,O-Z)
      DATA ONE,TWO/1.D0,2.D0/
      H=(EPS0*U)**(ONE/(M+TWO))
9000  RETURN
      END
```

```
      SUBROUTINE FSOL(U,F,DERF,XI,M,PI,FCT,W,ETA,EPSO)
      IMPLICIT REAL*8(A-H,O-Z)
      EXTERNAL PI,FCT,W
      DATA ZERO,ONE,TWO,THREE/0.0D0,1.0D0,2.0D0,3.0D0/
C                              THIS IS THE VALUE OF THE FUNCTION
C                              FBIG-FSMALL AND ITS DERIVATIVE.
C
C                              THE INDEPENT VARIABLE IS U
C                              COMPUTE FBIG
C
      CALL UTOH(U,HU,M,EPSO)
C                              HU IS H EXPRESSED IN FUNCTION OF U.
      X1=XI+ETA*HU
      FMPL1=FCT(X1,M+1)
      FM1SQR=FMPL1*FMPL1
      XL=ZERO
      XU=ONE
      INTKND=1
      SUM1=TEGRAL(XL,XU,HU,INTKND,XI,M,PI,W)
      FBIG=FM1SQR*SUM1
C
C                              COMPUTE FSMALL
C
      FSMALL=HU/(U*U)
C
C                              VALUE OF F IS :
C
      F=FBIG-FSMALL
C
C                              COMPUTE DFBIG
      FMPL2=FCT(X1,M+2)
      SUM2=TEGRAL(XL,XU,HU,1,XI,M,PI,W)
      SUM3=TEGRAL(XL,XU,HU,2,XI,M,PI,W)
       DHDU=HU/((M+2.D0)*U)
      DFBIG=(TWO*ETA*FMPL1*FMPL2*SUM2 +FM1SQR*SUM3)*DHDU
```

```
C
C                              COMPTE DFSMAL
C
C
C                              DHDU IS THE DERIVATIVE OF H VS U.
        COEF=-(TWO*M+THREE)/(M+TWO)
        DFSMAL=COEF*HU/(U*U*U)
       DERF=DFBIG-DFSMAL
9000   RETURN
       END

      SUBROUTINE HINIT(HINI,UINIT,XI,M,PI,FCT,W,ETA,EPSO)
      IMPLICIT REAL *8 (A-H,O-Z)
      EXTERNAL PI,FCT,W
      DATA ZERO,ONE,TWO,THREE/0.D0,1.D0,2.D0,3.D0/
10    CONTINUE
      XK=EPSO/FCT(XI,M+1)
      EXP1=ONE/(TWO*M+THREE)
      XK=(XK*XK)**(ONE/(TWO*M+THREE))
      XL=ZERO
      XU=ONE
C
C                              HU IS NOT USED WHEN INTKND = 3
C
      INTKND=3
      HU=0.D0
      WTSUM=W(XI,0)*TEGRAL(XL,XU,HU,INTKND,XI,M,PI,W)
      HINI=(ONE/WTSUM)**(EXP1)*XK
C
C                              INITIALIZE U
C
      F=FCT(XI,M+1)*DSQRT(WTSUM)
      F=DABS(F)
      UINIT=(ONE/F)*(EPSO/F)**EXP1
      RETURN
      END
```

```
      DOUBLE PRECISION FUNCTION TEGRAL(XL,XU,HU,INTKND,XI,M,PI,W)
      IMPLICIT REAL *8(A-H,O-Z)
      EXTERNAL PI,FCT,W
      A=.5D0*(XU+XL)
      B=XU-XL
      C=.4975936099985106 8D0*B
      Z=FCRIT(A+C,HU,INTKND,XI,M,PI,W)+FCRIT(A-C,HU,INTKND,XI,M,PI,W)
      Y=.6170614899993599 8D-2*Z
      C=.4873642779856547 5D0*B
      Z=FCRIT(A+C,HU,INTKND,XI,M,PI,W)+FCRIT(A-C,HU,INTKND,XI,M,PI,W)
      Y=Y+.1426569431446683 2D-1*Z
      C=.4691372760013663 8D0*B
      Z=FCRIT(A+C,HU,INTKND,XI,M,PI,W)+FCRIT(A-C,HU,INTKND,XI,M,PI,W)
      Y=Y+.2213871940870990 3D-1*Z
      C=.4432077635022005 2D0*B
      Z=FCRIT(A+C,HU,INTKND,XI,M,PI,W)+FCRIT(A-C,HU,INTKND,XI,M,PI,W)
      Y=Y+.2964929245771839 0D-1*Z
      C=.4100009929869514 6D0*B
      Z=FCRIT(A+C,HU,INTKND,XI,M,PI,W)+FCRIT(A-C,HU,INTKND,XI,M,PI,W)
      Y=Y+.3667324070554015 3D-1*Z
      C=.3700620957892771 8D0*B
      Z=FCRIT(A+C,HU,INTKND,XI,M,PI,W)+FCRIT(A-C,HU,INTKND,XI,M,PI,W)
      Y=Y+.4309508076597663 8D-1*Z
      C=.3240468259684877 8D0*B
      Z=FCRIT(A+C,HU,INTKND,XI,M,PI,W)+FCRIT(A-C,HU,INTKND,XI,M,PI,W)
      Y=Y+.4880932605205694 4D-1*Z
      C=.2727107356944197 7D0*B
      Z=FCRIT(A+C,HU,INTKND,XI,M,PI,W)+FCRIT(A-C,HU,INTKND,XI,M,PI,W)
      Y=Y+.5372213505798281 7D-1*Z
      C=.2168967538130225 7D0*B
      Z=FCRIT(A+C,HU,INTKND,XI,M,PI,W)+FCRIT(A-C,HU,INTKND,XI,M,PI,W)
      Y=Y+.5775283402686280 1D-1*Z
      C=.1575213398480816 9D0*B
      Z=FCRIT(A+C,HU,INTKND,XI,M,PI,W)+FCRIT(A-C,HU,INTKND,XI,M,PI,W)
      Y=Y+.6083523646390169 6D-1*Z
```

```
      C=.9555943373680815D-1*B
      Z=FCRIT(A+C,HU,INTKND,XI,M,PI,W)+FCRIT(A-C,HU,INTKND,XI,M,PI,W)
      Y=Y+.6291872817341414 8D-1*Z
      C=.3202844643130281 3D-1*B
      Z=FCRIT(A+C,HU,INTKND,XI,M,PI,W)+FCRIT(A-C,HU,INTKND,XI,M,PI,W)
      TEGRAL=B*(Y+.6396909767337607 8D-1*Z)
      RETURN
      END

            FUNCTION FCRIT(R,HU,INTKND,XI,M,PI,W)
            IMPLICIT REAL *8 (A-H,O-Z)
            EXTERNAL PI,FCT,W
C                                     THIS DEFINES THE FUNCION UNDER TH
C                                     SIGN
C                           R IS THE VARIABLE AND TAKES VALUES
C                           BETWEEN 0 AND 1
C                           INTKND DEPENDS ON THE KIND OF INTEGRAL
C                           INTKND = 1,2 OR 2.
C
      P1=PI(R,M+1)
      PISQAR=P1*P1
      XV=XI + R * HU
C
C                           HU IS THE VARIABLE TO FIND. IT IS A CHANGE
C                           OF VARIABLE FOR H.
C
      GO TO (1,2,3),INTKND
1     FCRIT=PISQAR * W(XV,0)
      GO TO 9000
C                           IN W(XV,N) N=ORDER OF THE DERIVATIVE.
2     FCRIT=PISQAR*R*W(XV,1)
      GO TO 9000
3     FCRIT=PISQAR
9000  RETURN
      END
```

```
      SUBROUTINE INALFA(ROOTS,ITYPE)
      IMPLICIT REAL*8(A-H,O-Z)
      DIMENSION ROOTS(10,10)
      DATA ZERO/0.D0/,ONE/1.D0/,TWO/2.D0/,THREE/3.D0/,FOUR/4.D0/,
     & FIVE/5.D0/
C
C     FIRST INDEX IS THE DEGREE OF THE POLYNOMIAL.
C     ROOTSS ARE THE ROOTS OF THE POLYNOMIAL OF TYPE ITYPE
C     ITYPE=1 IS TSCHEBYCHEFF
C
      PIDIV4=DATAN(ONE)
      DO 10 N=1,10
      DO 10 I=1,N
      X=(ONE+TWO*(I-1))*PIDIV4*TWO/N
      ROOTS(N,I)=(DCOS(X)+ONE)/TWO
      IF(MOD(N,2).EQ.1.AND.(I-1).EQ.(N-1)/2)ROOTS(N,I)=.5D0
10    CONTINUE
C
9000  RETURN
      END

      DOUBLE PRECISION FUNCTION PI(XV,M)
      IMPLICIT REAL*8(A-H,O-Z)
      COMMON /ALPHA/ALPHA(10,10)
      IF(M.EQ.0)PI=1.D0
      IF(M.EQ.0)GO TO 9000
      PI=1.D0
      MFACT=1
      DO 10 I=1,M
      PI=PI * (XV - ALPHA(M,I))
      MFACT=MFACT*I
10    CONTINUE
      PI=PI/MFACT
9000  RETURN
      END
```

```
      DOUBLE PRECISION FUNCTION W(XV,NTYPE)
      IMPLICIT REAL*8(A-H,O-Z)
      W=1.DO
C
C                              NTYPE REPRESENTS THE DEGREE OF DERIVATIVE
C
      IF(NTYPE.EQ.1)W=0.DO
      RETURN
      END

      DOUBLE PRECISION FUNCTION FCT(XV,M)
      IMPLICIT REAL*8(A-H,O-Z)
C
C     FUNCTION Y(X)=1/X
C
      IF(M.EQ.0)FCT=1.DO/XV
      IF(M.EQ.0)GO TO 9000
      XPOW=XV
      MFACT=1
      DO 10 I=1,M
      XPOW=XPOW * XV
      MFACT=MFACT * (-I)
10    CONTINUE
      FCT=DFLOAT(MFACT)/XPOW
9000  RETURN
      END
```

A.2 Distribution of Floating Point Numbers Outside the Interval [1/β, 1]

For the sine and exponential functions, the range cannot be reduced easily to [1/β, 1]. If we assume that the range has been reduced to [0,1], the arguments have the form:

$$X = x \cdot \beta^e$$

The distribution of e is problem dependent. For some problems, e may have a very narrow range of possible values while in another application there may be wide variations in the exponent e. The only thing that is known is the distribution $1/(x \ln \beta)$ of the mantissas x.

Nothing general is known about the distribution of the exponential in floating point numbers [SWE65] and consequently the only choice is to assume that the e's have a uniform distribution. Representing x between 0 and 1 requires values of e varying between 0 and a large negative number which we will take to be $-\infty$ for all practical purposes. What, then, is the probability density function of the argument X?

For $X \in [\frac{1}{\beta}, 1]$ $e = 0$

For $X \in [\frac{1}{\beta^2}, \frac{1}{\beta}]$ $e = -1$

For $X \in [\frac{1}{\beta^3}, \frac{1}{\beta 2}]$ $e = -2$

For $X \in [\frac{1}{\beta^{i+1}}, \frac{1}{\beta^i}]$ $e = -i$

Within each one of these intervals the probability density of the mantissas x is $1/(x \ln\beta)$. Consequently, the probability density of X for X $[\frac{1}{\beta^{i+1}}, \frac{1}{\beta^{i}}]$ is

$$\frac{1}{x \ln\beta} \frac{\beta - 1}{\beta^{i+1}}$$

where $\frac{\beta - 1}{\beta^{i+1}}$ is the uniform density of β^{e} in the interval $[\frac{1}{\beta^{i+1}}, \frac{1}{\beta^{i}}]$.

The probability density function for numbers between 0 and 1 is sketched in Fig. A.1. This is the distribution that must be used for solving the NLP problem in the case of $f(x) = e^{x}$ or $f(x) = \sin(\frac{\pi}{4} x)$.

It is interesting to note that, if several polynomial degrees are chosen, it is better, in this case, to put the lowest degree polynomials close to 1 and the highest degree polynomials close to 0. X is found more often at the right of the interval [0, 1]. The probability of finding X close to 0 gets smaller as X tends to 0.

The cumulative probability between $\frac{1}{\beta^{i}}$ and $\frac{1}{\beta^{i-1}}$ is:

$$\frac{\beta - 1}{\ln \beta \cdot \beta^{i}} \int_{\frac{1}{\beta}}^{1} \frac{dx}{x} = \frac{\beta-1}{\beta^{i}}$$

which tends towards 0 as i increases. The average becomes smaller as $X \to 0$.

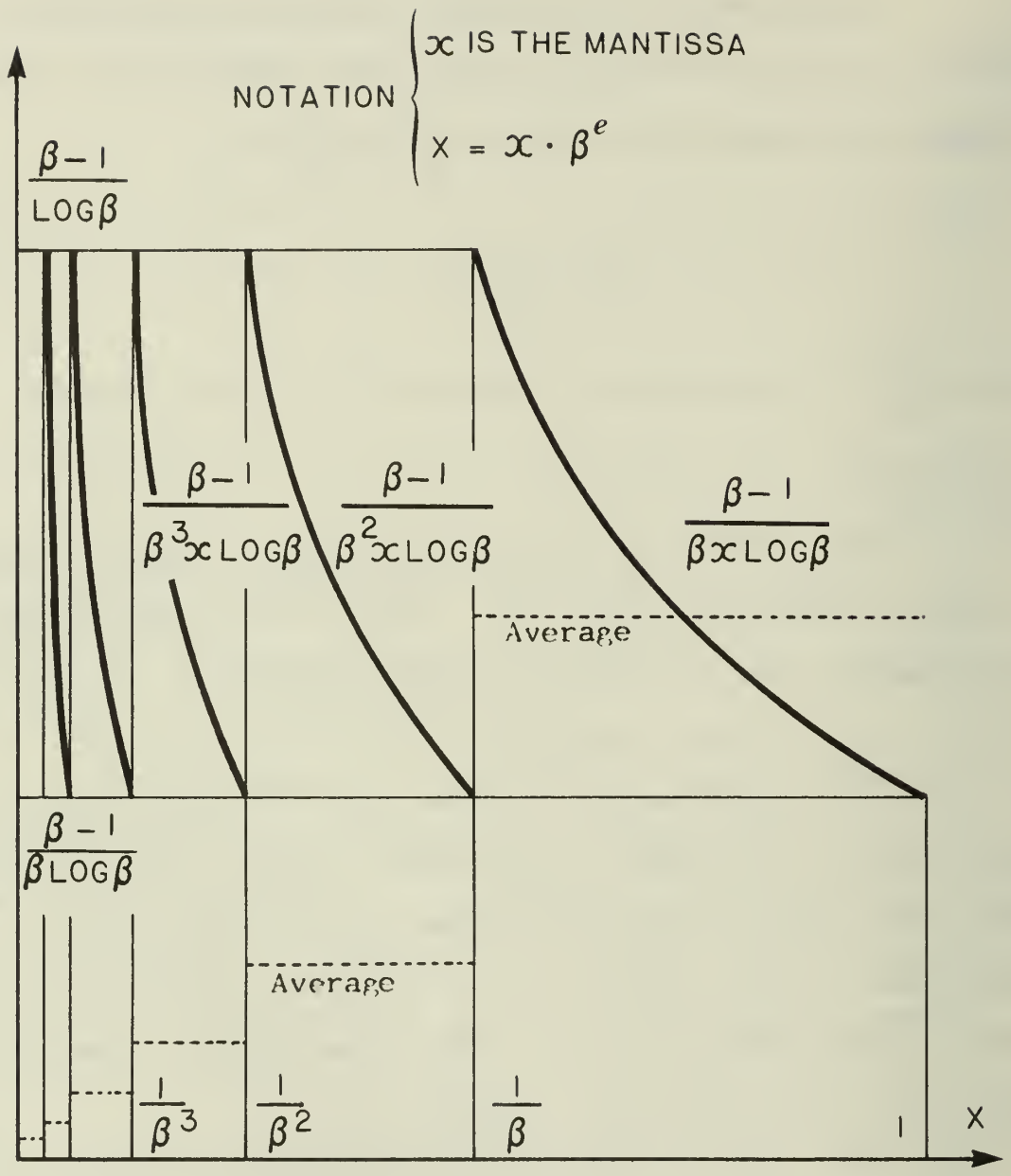


Figure A.1 Probability Density Function of Floating Point Numbers between 0 and 1

A.3 Limit on ROM Savings Coefficient When $n \to \infty$

We have seen in Section 3.3.2 that the "ROM Savings Coefficient" defined as:

$$\mathcal{V}_{(a,b)} = \frac{g(a)}{b - a} \int_a^b \frac{dx}{g(x)}$$

where

$$g(x) = \frac{1}{\left(f^{(m+1)}(x)\right)^{\frac{2}{(2m+3)}}}$$

(here $w(x) = 1$) was obtained as the ratio:

$$\mathcal{V}_{(a,b)} = \frac{\text{number of breakpoints in [a,b] using variable intervals}}{\text{number of breakpoints in [a,b] using uniform intervals}}$$

The smaller $\mathcal{V}_{(a,b)}$ is, the greater the savings in ROM.

We are concerned here with whether or not this ratio increases as the polynomial degree increases. In other words: as $m \to \infty$ is it worth using variable intervals to save memory?

We already know that, to first approximation, $g(x)$ does not depend on n. We will consequently find an approximate expression for $g(x)$ and look at the limit of the integral when m, the degree of the polynomial tends to ∞. As in Section 7 we will consider Type 1 and Type 2 functions.

Type 1 Function:

$$f^{(m+1)}(x) = O\left(\frac{C_1^{m+1}(m+1)!}{x^{m+1}}\right)$$

We then have:

$$\mathcal{V}_1(a,b) = \frac{1}{b-a}\int_a^b \frac{g(a)}{g(x)}\,dx = \frac{1}{b-a}\int_a^b \left[\frac{f^{(m+1)}(x)}{f^{(m+1)}(a)}\right]^{\frac{2}{2m+3}} dx$$

for a type 1 function, $\mathcal{V}(a,b)$ becomes:

$$\mathcal{V}(a,b) = \frac{1}{b-a}\int_a^b \left(\frac{a^{m+1}}{x^{m+1}}\right)^{\frac{2}{2m+3}} dx$$

as $m \to \infty$

$$\mathcal{V}(a,b) \approx \frac{1}{b-a}\int_a^b \frac{a}{x}\,dx$$

or

$$\mathcal{V}_1(a,b) = \frac{a}{b-a}(\ln b - \ln a)$$

In percent, the ROM usage is then:

$$100\left(1 - \frac{a}{b-a}(\ln b - \ln a)\right)$$

Example: If a = .5, b = 1

$$\mathcal{V}_1(a,b) = \frac{.5}{.5}\,[\,\ln 1. - \ln .5]$$

$$\mathcal{V}_1(a,b) = .69$$

which means that asymptotically the savings would be of 31%.

Type 2 Functions:

We will take the exponential function as an example.

$$f^{(m+1)}(x) = e^x$$

$$\mathcal{V}_2(a,b) = \frac{1}{b-a}\int_a^b \left(\frac{e^x}{e^a}\right)^{\frac{2}{2m+3}} dx$$

as $m \to \infty$

$$\mathcal{V}_2(a,b) \approx \frac{1}{(b-a)e^{a/m}}\int_a^b e^{x/m}\,dx$$

evaluating the definite integral yields

$$\mathcal{V}_2(a,b) \approx \frac{m}{(b-a)}\,(e^{(b-a)/m} - 1)$$

and as $m \to \infty$ $(b-a)/m \to 0$, and using $e^x - 1 = x + \frac{x^2}{2!} + \ldots$

$$\mathcal{V}_2(a,b) = \frac{m}{(b-a)}\left(\frac{b-a}{m} + \frac{1}{2!}\left(\frac{b-a}{m}\right)^2 + \ldots\right)$$

$$\mathcal{V}_2(a,b) = 1 + \frac{b-a}{2!m} + \frac{(b-a)^2}{3!m^2} + \ldots$$

Therefore

$$\mathcal{V}_2(a,b) \longrightarrow 1 \quad \text{when } m \to \infty$$

□

Consequently for $f(x) = e^x$ as $m \to \infty$ the ROM savings tend to be null when using variable vs uniform intervals.

It is possible to show that the same thing would occur for the functions sin and cos:

choosing $f(x) = \cos x$, for example

$$\mathcal{V}_{(a,b)} = \frac{1}{b-a}\int_a^b \left(\frac{\cos x^{(m+1)}}{\cos a^{(m+1)}}\right)^{\frac{2}{2m+3}} dx$$

$[(\cos n)^{m+1}]^2$ is either $(\cos x)^2$ or $(\sin x)^2$, so $\mathcal{V}_{(a,b)}$ can be expressed as

$$\mathcal{V}_{(a,b)} = \frac{1}{(b-a)(\cos a)^{1/m}}\int_a^b (\sin x)^{\frac{1}{m}} dx$$

or

$$\mathcal{V}_{(a,b)} = \frac{1}{(b-a)(\cos a)^{1/m}}\int_a^b |\cos x|^{\frac{1}{m}} dx$$

We know that

$$\int_a^b \sin^{r-1}x.dx = \int_a^b \cos^{r-1}x.dx = \frac{\sqrt{\pi}\,\Gamma(r/2)}{2\;\Gamma(r/2+1/2)} \quad ,r > 0$$

applying this formula with $r = \frac{1}{m} + 1$ and having $m \to \infty$ yields:

$$\mathcal{V}_{(0,\pi/2)} = \frac{1}{\frac{\pi}{2}-0} \cdot \frac{\sqrt{\pi}\,\Gamma(1/2)}{2\,\Gamma(1)}$$

but

$$\Gamma(1/2) = \sqrt{\pi} \quad \text{and} \quad \Gamma(1) = 1$$

consequently

$\mathcal{V}_{(0,\pi/2)} = 1$ the ROM savings is null.

We conjecture here that for type 2 functions, the ROM savings limit is null when the polynomial degrees tend to infinity.

VITA

Gilles Henri Garcia was born in France in 1947. He received degrees in Electrical Engineering from the University of Toulouse in 1970 and from the University of Paris in 1972. With the aid of a Fulbright Grant, he came to the University of Illinois at Urbana-Champaign, where he received his M.S. and Ph.D. in Computer Science in 1976 and 1978, respectively.

From 1973 to 1977, Mr. Garcia was employed as a graduate teaching and research assistant by the Department of Computer Science and the Institute of Aviation at the University of Illinois. He has co-authored a paper in IEEE Transactions in Computers. From 1975 to 1977, Mr. Garcia was also associated with the Marketing Department of the Illinois Bell Telephone Company in Chicago as a consultant. He is a member of ACM and of the Société des Ingenieurs de l'Ecole Supérieure d'Electricité. Mr. Garcia is currently employed by Schlumberger Well Services, Log Analysis Software, in Houston, Texas.

BIBLIOGRAPHIC DATA SHEET	1. Report No. UIUCDCS-R-78-922	2.	3. Recipient's Accession No.

Title and Subtitle	5. Report Date
MINIMUM MEAN RUNNING TIME FUNCTION GENERATION USING READ ONLY MEMORY	Dec. 1978
	6.
Author(s)	8. Performing Organization Rept. No. UIUCDCS-R-78-922
Gilles Henri Garcia	
Performing Organization Name and Address	10. Project/Task/Work Unit No.
Department of Computer Science University of Illinois Urbana-Champaign, IL	11. Contract/Grant No.
Sponsoring Organization Name and Address	13. Type of Report & Period Covered
Department of Computer Science University of Illinois	Ph.D. Thesis
	14.

Supplementary Notes

Abstracts

This thesis presents a method for generating functions using a minimum mean running time polynomial approximation. Although the method can be used for software library routines, the work focuses on hardware implementations. With the advent of LSI, Read Only Memories may be used to store many constants very economically. In addition, large multipliers are now available which multiply two n-bit number in O(n) or O(logn) time. In this discussion, multiplication is considered as the basic operator. The method consists of splitting the interval [a,b] into several large partitions. Within each large partition, the function f(x) is evaluated in a large number of small subintervals using an approximating polynomial of very low degree. The coefficient of the polynomials are stored in ROMs.

Key Words and Document Analysis. 17a. Descriptors

Polynomial approximation
Read Only Memories

Identifiers/Open-Ended Terms

COSATI Field/Group

Availability Statement	19. Security Class (This Report)	21. No. of Pages
Unlimited	UNCLASSIFIED	227
	20. Security Class (This Page) UNCLASSIFIED	22. Price